Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa magisterska

Projekt języka programowania wspierającego przetwarzanie rozproszone na platformach heterogenicznych.

Design of a programming language with support for distributed computing on heterogenous platforms.

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

Kraków, 2015

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 opiekunowi pracy za wsparcie merytoryczne oraz dobre rady edytorskie pomocne w tworzeniu pracy.

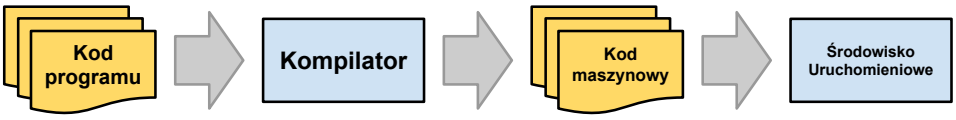
Spis treści

1. Wstęp

Tematem pracy jest projekt i implementacja języka programowania wspierającego *przetwarzanie współbieżne* i umożliwiającego tworzenie *systemów rozproszonych* działających na *platformach heterogenicznych*.

Przetwarzanie współbieżne polega na podziale obliczeń na wiele procesów działających jednocześnie i konkurujących ze sobą o dostęp do ograniczonej ilości zasobów [1]. Procesy te mogą zostać rozproszone na wiele fizycznych maszyn, zachowując jednocześnie komunikację pomiędzy nimi, tworząc tym samym jeden koherentny system rozproszony [2].

Projektowany język programowania powinien więc udostępniać przejrzystą i ogólną notację umożliwiającą definiowanie komunikujących się procesów, jednocześnie pozostając językiem ogólnego przeznaczenia. Dodatkowym wymogiem jest prostota przy zachowaniu ekspresywności - język ten powinien być zbudowany w oparciu o niewielką liczbę ortogonalnych, dobrze współgrających ze sobą mechanizmów, które pozwalają na implementację szerokiej gamy funkcjonalności [3].



Rysunek 1.1: Schemat interakcji poszczególnych elementów języka.

W tym celu wymagane jest stworzenie kompilatora, czyli programu transformującego kod źródłowy języka programowania na format możliwy do uruchomienia w pewnym środowisku uruchomieniowym (ang. *runtime system*). Na środowisko takie zazwyczaj składa się zestaw podstawowych procedur wspólnych i niezbędnych do działania każdego programu. Schemat interakcji poszczególnych elementów projektu zaprezentowano na schemacie 1.1.

Ostatnim wymogiem postawionym przed projektowanym językiem, jest wyjście naprzeciw licznym problemom występującym w systemach rozproszonych, w szczególności problemowi *heterogeniczności*, co umotywowano w następnej sekcji.

1.1. Motywacja pracy

Tworzenie systemów rozproszonych jest zadaniem bardzo trudnym i wymaga wykorzystania specjalnie do tego przeznaczonych narzędzi - języków programowania, systemów bazodanowych i infrastruktury sieciowej. Na przestrzeni lat zidentyfikowano wiele kluczowych problemów manifestujących się we wszystkich systemach rozproszonych niezależnie od ich przeznaczenia. Tanenbaum oraz Van Steen w [2] wymieniają następujące problemy:

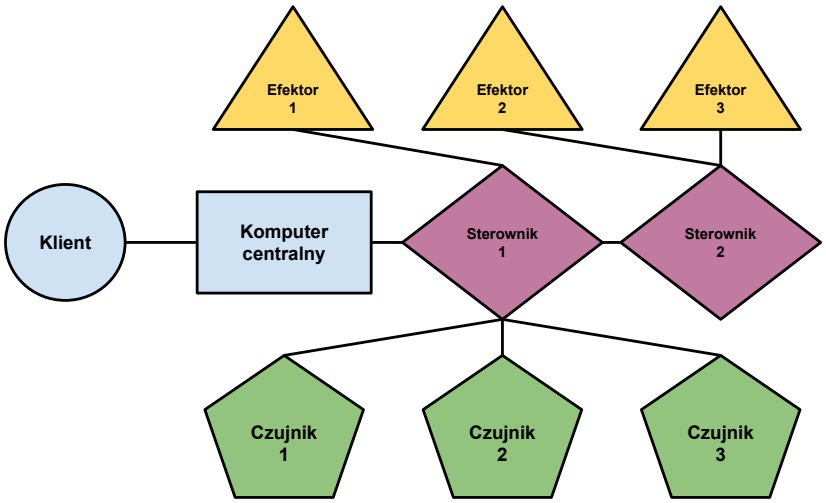
- **dostępność** - systemy rozproszone działają zwykle na wielu odrębnych maszynach, istotnym jest więc zachowanie dostępu do wspólnych zasobów z każdej części systemu,
- **przezroczystość dystrybucji** - istotnym jest również ukrycie fizycznego rozproszenia procesów i zasobów, dzięki czemu system rozproszony z zewnątrz stanowi jedną, koherentną całość,
- **otwartość** - polega na standaryzacji komunikacji pomiędzy poszczególnymi częściami systemu, dzięki czemu możliwe jest dodawanie nowych elementów bez ingerencji w pozostałe,
- **skalowalność** - systemy rozproszone powinny umożliwiać płynną zmianę ilości zasobów, być dostępne dla użytkowników z wielu lokacji geograficznych oraz umożliwiać łatwe zarządzanie niezależnie od ich rozmiaru.

Warto zauważyć, iż *skalowalność* jest przywoływana w folklorze programistycznym nieproporcjonalnie często, natomiast, istnieje wiele równie trudnych problemów, którym przeznacza się znacznie mniej uwagi, takich jak:

- **bezpieczeństwo systemu** - polega na kontroli dostępu do zasobów; w zależności od przeznaczenia systemu rozproszonego, jest najważniejszym aspektem jego budowy,
- **odporność na błędy** - polega na reagowaniu na zmiany (w szczególności wystąpienie błędów) zachodzące w systemie i podejmowaniu odpowiednich akcji w razie ich wystąpienia,
- **heterogeniczność** - polega na zróżnicowaniu platform sprzętowych wchodzących w skład fizycznej części systemu rozproszonego, a także poszczególnych logicznych części systemu.

Heterogeniczność jest problemem szczególnie trudnym, który powoli nabiera znaczenia wraz z pojawieniem się inicjatyw takich jak **Internet Rzeczy** (ang. *Internet of Things*) [4], gdzie systemy rozproszone zbudowane są z dużej ilości bardzo zróżnicowanych maszyn. Maszyny te cechują się różną architekturą sprzętową, ilością zasobów, a także przeznaczeniem i funkcjonalnościami, które realizują i umożliwiają.

Na schemacie 1.2 przedstawiony został przykład heterogeniczności platformy sprzętowej w kontekście Internetu Rzeczy. Klient, korzystając z centralnego komputera, uzyskuje dostęp do danych sensorycznych pochodzących z szerokiej gamy różnych czujników i bazując na ich wartości jest w stanie zmieniać zachowanie równie zróżnicowanych efektorów. Całość odbywa się za pośrednictwem dedykowanych sterowników, ułatwiających skalowanie systemu.

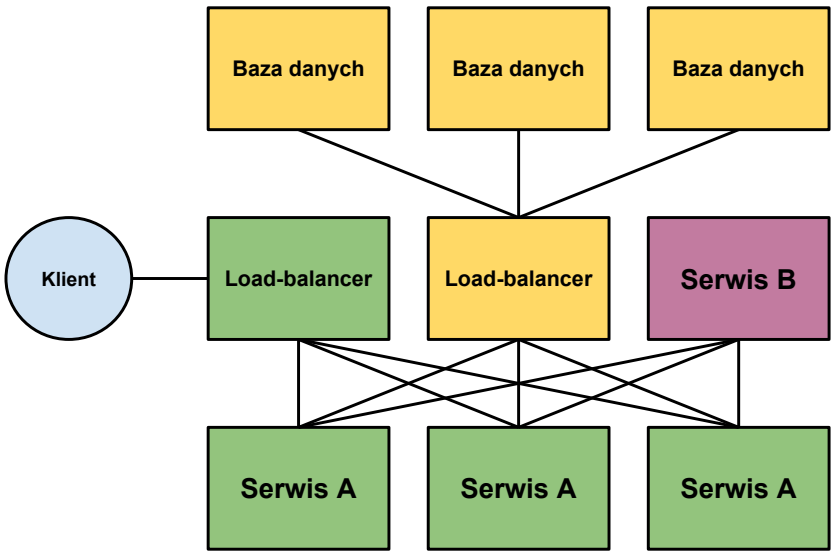


Rysunek 1.2: Przykład systemu opartego o heterogeniczną platformę sprzętową.

Każdy element takiego systemu, oznaczony na schemacie różnym kształtem oraz kolorem, reprezentuje maszynę udostępniającą różne zasoby i posiadającą różną fizyczną konstrukcję. Poszczególne elementy często działają w niekompatybilny sposób, w związku z czym wymagane jest wykorzystanie dedykowanych pośredników, których jedynym zadaniem jest *homogenizacja* systemu.

Problem heterogeniczności dotyka również systemów rozproszonych, które działają na platformach homogenicznych, gdzie fizyczne maszyny są do siebie bardzo zbliżone, a często są komputerami ogólnego przeznaczenia. Przykład takiego systemu, zbudowanego w oparciu o zdobywającą popularność architekturę mikroserwisową [5], został zawarty na schemacie 1.3.

Użytkownik systemu łączy się z głównym serwisem, który następnie komunikuje się z innymi serwisami, realizującymi wymagane przezeń zadania. W celu poprawienia *odporności na błędy* takiego systemu, w strategicznych miejscach umieszczono serwery zarządzające ruchem (ang. *load-balancer*), których zadaniem, analogicznie do przykładu ze schematu 1.2, jest *homogenizacja* systemu.



Rysunek 1.3: Przykład systemu heterogenicznego niezależnie od platformy sprzętowej.

W pierszym przypadku heterogeniczność wynika ze zróżnicowania maszyn należących do platformy sprzętowej, natomiast w drugim wynika ona z istnienia mikroserwisów, które realizują pojedyncze, konretnie sprecyzowane funkcjonalności. W obu przypadkach, heterogeniczność systemu prowadzi do powstania innych problemów, takich jak skalowalność i odporność na błędy, oraz konieczności wykorzystania dodatkowych elementów mających im zaradzić.

Często, sytuacja ta wynika z nieadekwatności narzędzi (w szczególności języków programowania) wykorzystanych do tworzenia systemu. Popularne języki programowania dążą do osiągnięcia **niezależności od platformy** (ang. *platform independence*) stosując maszyny wirtualne i inne techniki mające na celu homogenizację platformy sprzętowej, kiedy w rzeczywistości osiągają **ignorancję platformy** nie umożliwiając refleksji na jej temat.

Jako alternatywę dla osiągnięcia niezależności od platformy, niniejsza praca wprowadza termin **świadomości platformy** (ang. *platform awareness*), czyli dążenia do udostępnienia wiedzy o strukturze budowanego systemu rozproszonego oraz platformy sprzętowej, na której działa, i umożliwienia refleksji na jej podstawie. Zaprezentowany w dalszej części pracy język programowania, roboczo zwany `FOOF` [1], ma być uosobieniem ideologii świadomości platformy.

1.2. Zawartość pracy

– list what is found where in the thesis

[1] Nazwa pochodzi od difluorku ditlenu, niezwykle reaktywnego, dysruptywnego i niebezpiecznego związku chemicznego, który nie ma zastosowania.

2. Język FOOF

Niniejszy rozdział szczegółowo opisuje projekt języka programowania FOOF począwszy od podstawowych typów danych, przez notację funkcji, kontynuacji i procesów, kończąc na zaawansowanych mechanizmach języka, takich jak przetwarzanie wiedzy i wbudowany system makr. W dodatku A zawarto formalny opis gramatyki języka, natomiast w dodatku B zamieszczono kilka przykładowych programów.

Język FOOF został zaprojektowany bazując na cennych wskazówkach przedstawionych przez John'a Backus'a w wykładzie wygłoszonym przez niego podczas odbierania Nagrody Turing'a w 1977 roku [3]. Wskazówki te są ponadczasowe i stanowią dobrą podstawę do tworzenia języków programowania, a w dużym skrócie sprowadzają się do następujących punktów:

- **prostota lecz nie surowość** (ang. *simplicity, not crudeness*) - języki programowania powinny cechować się prostotą, lecz nie ograniczać ekspresywności programisty przez brak możliwości zrealizowania pewnych funkcjonalności, a co za tym idzie:

- **ortogonalne funkcjonalności** (ang. *orthogonal features*) - język programowania powinien składać się z niewielkiej liczby dobrze zdefiniowanych i dobrze współgrających mechanizmów, za pomocą których programista jest w stanie łatwo zbudować wszelkie inne potrzebne funkcjonalności.

Oczywiście, zasady te nie są wystarczające do stworzenia funkcjonalnego języka programowania, dlatego kierowano się także **pragmatyzmem**, który w kontekście projektowania języków programowania sprowadza się do podejmowania kompromisów, pomiędzy *matematyczną czystością* a faktyczną użytecznością potencjalnych funkcjonalności dostarczanych przez język. Podejście to zostało szczegółowo opisane w [6].

Ze względu na podobne zasady, którymi kierowano się podczas projektowania, język FOOF przypomina pod względem składniowym i semantycznym odpowiednio języki **Scheme** (opisany szczegółowo w [7]) oraz **Standard ML** [8]. Natomiast, cechami odróżniającymi FOOF od tych języków są: wsparcie dla programowania współbieżnego oraz wykorzystanie inżynierii wiedzy w celu osiągnięcia *świadomości platformy* i rozwiązania problemu heterogeniczności systemów rozproszonych.

2.1. Podstawowe typy danych

Listing 1 prezentuje proste typy danych dostępne w języku FOOF; są to podstawowe elementy budulcowe programów, które mają swoją reprezentację literałową.

```
23.5
symbol
:symbol
"ciąg znaków"
(1 2 3)
[1 2 3]
{:a 1 :b 2}
```

Listing 1: Podstawowe typy danych dostępne w języku FOOF.

Typy te to w kolejności: liczby, symbole, słowa kluczowe i ciągi znaków tekstowych, stanowiące wspólnie klasę wartości atomowych oraz listy pojedynczo-wiązane, wektory i mapy asocjacyjne. Każdy nieatomowy typ danych składa się z określonej liczby podwartości, które mogą być atomowe, lub nieatomowe. Semantyka każdego wymienionego typu danych jest zgodna z opisem zawartym w [7].

Jako, że język FOOF jest dialektem języka Lisp, programy kodowane są homoikonicznie przez opisane powyżej typy danych - stosowana jest notacja **S-wyrażeń**, która została wprowadzona w [9]. Notacja ta rozmywa granicę pomiędzy programami a danymi, pozwalając programom na manipulację, budowę i transformację innych programów.

Homoikoniczność i notację S-wyrażeń wykorzystano w wielu innych mechanizmach dostępnych w języku, które zostały opisane w dalszej części niniejszego rozdziału, w szczególności w implementacji systemu makr pozwalających na rozszerzenie składni języka.

2.2. Funkcje

Pierwszym złożonym typem danych, który nie ma reprezentacji literałowej w języku FOOF są funkcje. Funkcje są obiektami pierwszej klasy, to znaczy, po stworzeniu podczas działania programu, mogą być wykorzystywane tak jak każdy inny typ danych, a co za tym idzie, mogą być osadzane w listach, przekazywane do innych funkcji, a także z nich zwracane jako wynik obliczeń.

Funkcje zostały zaprojektowane w oparciu o **rachunek Lambda**, wprowadzony w 1933 roku przez Alonzo Church'a jako alternatywny model logiki i, następnie, prowadzenia obliczeń [10, 11]. Rachunek ten wprowadza pojęcie **wyrażenia lambda**, które jest ekwiwalentem jednoargumentowych funkcji obecnych języków programowania, oraz szereg zasad substytucji,

zwanych redukcjami, pozwalających na uproszczenie zagnieżdżonych wyrażeń lambda. Najważniejszą z wprowadzanych redukcji jest β**-redukcja**, która konceptualnie reprezentuje aplikacje funkcji z odpowiednimi argumentami i jednocześnie pozwala na prowadzenie obliczeń.

Zasady rachunku lambda są fundamentalnie bardzo nieskomplikowane, a mimo to pozwalają na ekspresję skomplikowanych idei, takich jak logika Bool'a, arytmetyka, struktury danych a także rekurencja. Na listingu 2 zawarto przykład realizacji logiki boola wraz z kilkoma operatorami logicznymi w czystym rachunku lambda.

```
1  TRUE := λx.λy.x
2  FALSE := λx.λy.y
3
4  AND := λp.λq.p q p
5  OR := λp.λq.p p q
6  NOT := λp.λa.λb.p b a
7
8  AND TRUE FALSE
9      ≡ (λp.λq.p q p) TRUE FALSE →β TRUE FALSE TRUE
10     ≡ (λx.λy.x) FALSE TRUE →β FALSE
```

Listing 2: Przykład implementacji wartości i operatorów logicznych jedynie za pomocą wyrażeń lambda.

Wartości logiczne kodowane są jako wyrażenia lambda konsumujące dwa argumenty i wybierające odpowiednio pierwszy z nich, dla logicznej wartości prawdy, lub drugi z nich, dla logicznej wartości fałszu. W podobny sposób kodowane są operatory logiczne, a wynik ich działania obliczany jest przez sukcesywne przeprowadzanie substytucji nazwy argumentu na jego wartość oraz redukowaniu otrzymanych wyrażeń za pomocą β-redukcji.

Warto zauważyć, że wyrażenia lambda można interpretować jako tak zwane **domknięcia leksykalne**, czyli tworzone podczas β-redukcji otaczającego wyrażenia pary funkcji i map asocjacyjnych odzwierciedlających wartości zmiennych, które występują w ciele domknięcia leksykalnego, a nie są przez nie wprowadzane. Domknięcia leksykalne pozwalają opóźnić substytucję nazw argumentów wyrażeń lambda na odpowiadające im wartości, dzięki czemu są łatwiejsze w implementacji [12].

Listing 3 pokazuje działanie domknięć leksykalnych w notacji języka FOOF.

```
(let* ((x 23)
       (foo (lambda () x)))
  (let ((x 5))
    (display (foo)))) ;; Wyświetla liczbę 23
```

Listing 3: Przykład ilustrujący działanie domknięć leksykalnych.

Funkcja `foo` zaprezentowana na listingu, korzysta z wartości **wolnej zmiennej** `x`, czyli takiej, której nie wprowadza w liście swoich argumentów. W dalszej części programu, funkcja `foo` pomimo lokalnej zmiany wartości zmiennej `x` poprawnie zwraca oryginalną jej wartość, ponieważ w momencie jej tworzenia wartość zmiennej `x` została zapisana razem z ciałem funkcji.

Często pojawiającym się problemem związanym z funkcjami wzorowanymi na rachunku lambda, jest tak zwany problem **funarg**, polegający na niepoprawnym działaniu programów, które zwracają funkcje jako wynik obliczeń, lub przekazują je jako argumenty innych funcji. Problem ten sprowadza się do niewłaściwego budowania domknięć leksykalnych, co może doprowadzić do przedwczesnego usunięcia wartości zmiennych wolnych. Został on poruszony w [13].

Kolejnym problemem towarzyszącym funkcjom zrealizowanym jako domknięcia leksykalne jest nietrywialna implementacja rekurencji, wynikająca z ustalonej kolejności wykonywania działań - tworzenie domknięcia leksykalnego funkcji rekurencyjnej jest uzależnione od jej uprzedniego istnienia, co prowadzi do sprzeczności.

Oryginalna praca wprowadzająca rachunek lambda w celu osiągnięcia rekurencji wykorzystuje rachunek kombinatorów [10], a w szczególności **kombinator Y**. Sposób działania tego kombinatora został szczegółowo opisany w [14], natomiast problem i propozycję implementacji rekurencji szerzej opisano w [15].

2.3. Kontynuacje

– describe the notion of a continuation [[16]]

– briefly describe CPS transformation and comment on code equivalence [[17]]

```
;; Styl aplikatywny:
(+ 2 (* 3 4))
```

```
;; Styl Continuation Passing:
(*& 3 4
    (lambda (v)
      (+& 2 v identity)))
```

Listing 4: Przykład konwersji *Continuation Passing Style*.

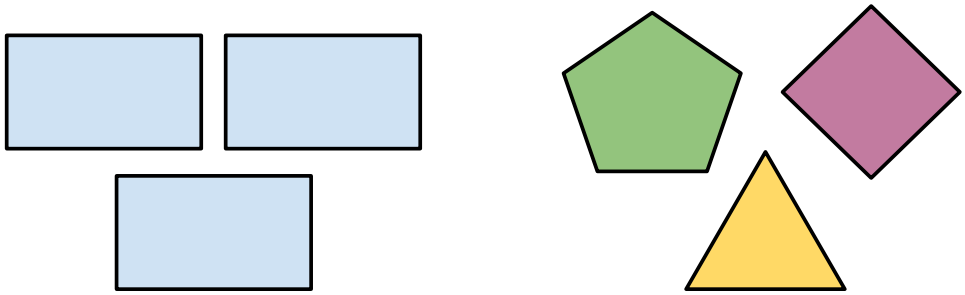
– list some popular primitives - letcc, call/cc [[7]] [[18]]

– hint at delimited control - shift & reset [[19]]

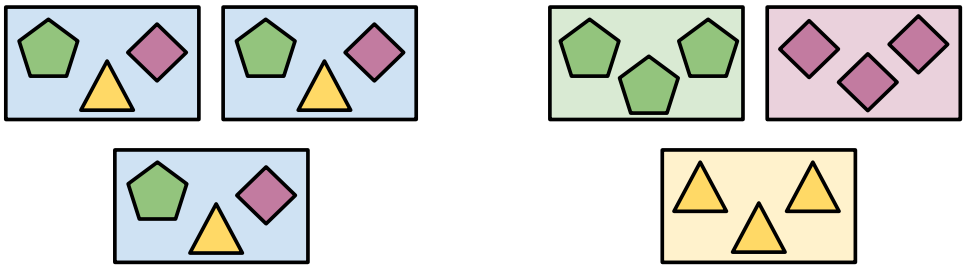
– describe exceptions via continuations - handle & raise

2.4. Przetwarzanie współbieżne i rozproszone

– briefly describe AMP vs SMP and contrast it with platform heterogeneity



Rysunek 2.1: Podstawowe różnice pomiędzy platformami homogenicznymi oraz heterogenicznymi.



Rysunek 2.2: Podstawowe różnice pomiędzy systemami asymetrycznymi i symetrycznymi.

– note that system doesn't need to run on a heterogenous platform to be heterogenous itself

– describe Actor Model [[20]] [[21]]

– describe actor model primitives [[20]]

```
(send (spawn (lambda ()
               (sleep 1000)
               (send (recv) 'message)))
      (self))

(equal? (recv) 'message)
```

Listing 5: Przykład wykorzystania prymitywnych operacji Modelu Aktorowego w języku.

- compare with Erlang [[22]]
- hint at processes implemented via continuations (trampolines)
- comment on extending this framework to add distribution

2.5. Reprezentacja wiedzy w języku

- describe use cases in the language
- describe various ways of knowledge representation [[23]] [[24]] [[25]]
- describe primitive operations

```
(signal! an-event)

(whenever set-of-conditions
  (lambda (_)
    (retract! some-fact)
    (assert! another-fact)))
```

Listing 6: Przykład wykorzystania prymitywnych operacji bazy wiedzy w języku.

- hint at using an RBS

2.6. Makra

- describe macros & macroexpansion

```
;; Przed makro-ekspansją:
(and 23 42)



(let ((x 23))
  (display x))


`(4 is ,(* 2 2))
```

```
;; Po makro-ekspansji:
(if 23
    42
    false)

((lambda (x)
   (display x))
 23)

(list '4 'is (* 2 2))
```

Listing 7: Przykład działania systemu makr w języku FOOF.

– note about the usage of quasiquote [[26]]

– hint at problems of hygiene & add code example [[27]]

```
(define-macro (unless c . b)
  `(if (not ,c)
       (begin ,@b)
       #void))

(let ((not identity))
  (unless #t
    (display "Hello world!")))
```

Listing 8: Przykład ilustrujący problem higieniczności systemu makr w języku Scheme.

– compare different styles of macro systems (syntax-rules & define-macro) [[7]]

– note that macros are not first-class [[28]]

– contrast macros with other techniques (fexprs) [[29]]

2.7. System modułowy

– describe the need for a module system [[30]]

```
(module (A)
  (define (foo x)
    (+ 23 x)))

(module (B a)
  (define (bar)
    (a.foo 5)))

(let ((b (B (A))))
  (display (b.bar))) ;; Wyświetla liczbę 28
```

Listing 9: Przykład wykorzystania systemu modułowego języka FOOF.

- describe structures - namespaces for definitions
- note about special structure access syntax - foo.bar
- describe modules - parameterized structures [[31]]
- describe units - runnable modules
- describe protocols - a set of capabilities of a module (?)
- hint at protocols & SOA connection ?
- note about all these primitives being macros
- contrast described system with SML [[8]]
- hint at problems of macros & modules coexisting [[30]]

3. Kompilator języka FOOF

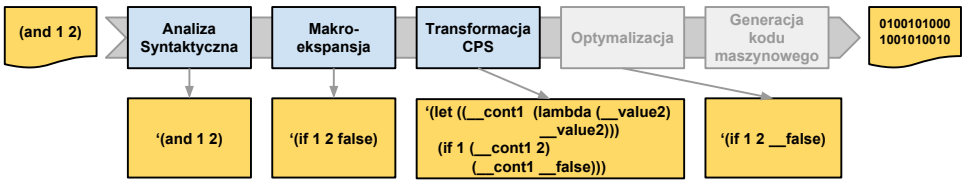
– mention technology selection & limitations (large project, little time) [[32]]

– mention possible bootstrapping

– briefly touch on the architecture [[33]]

– hint at using Scheme for the boring details (datatypes etc)

3.1. Architektura kompilatora



Rysunek 3.1: Schemat poszczególnych faz kompilacji i przykładowych danych będących wynikiem ich działania.

– list compilation phases [[33]] [[32]] [[17]]

– list which phases have been actually implemented

– list which phases have been skipped and say why (optimization, code-gen, parsing)

3.2. Parser

– briefly describe how Scheme praser works and what it produces [[7]] [[13]]

– hint at a possibility of replacing this with a PEG-based packrat [[34]] [[35]]

– note about special quasiquote syntax [[26]]

3.3. Makro-ekspansja

– describe macroexpantion phase

– describe why macroexpansion is hardcoded [[30]]

– list available macros

– show some examples of macro-expanded code

3.4. Obsługa Systemu Modułowego

– describe how modules are handled right now [[30]] [[31]]

– show some examples of macro-expanded structures & modules

– maby combine this with the previous section ?

– maby hint at special module access syntax (foo.bar.baz)

3.5. Transformacja *Continuation Passing Style*

– describe what CPS is [[17]] [[36]]

– describe in detail how to transform simple stuff

– describe in detail how to transform functions (recursion problems & crude solution via mutation [[37]], [[38]], [[14]])

– describe in detail how to handle exceptions

– describe in detail why this is useful (partial evaluation, constant folding etc) [[39]]

– hint at emitting calls to primitive functions `&yield-cont`, `&uproc-error-handler` etc

3.6. Generacja kodu

– describe how a subset of both Scheme and FOOF is emitted (contrast with Core Erlang) [[40]] [[41]]

– describe how Scheme is used for direct code execution

– hint at further development using LLVM [[**?**]]

– mention a requirement to perform closure conversion or lambda lifting [[12]]

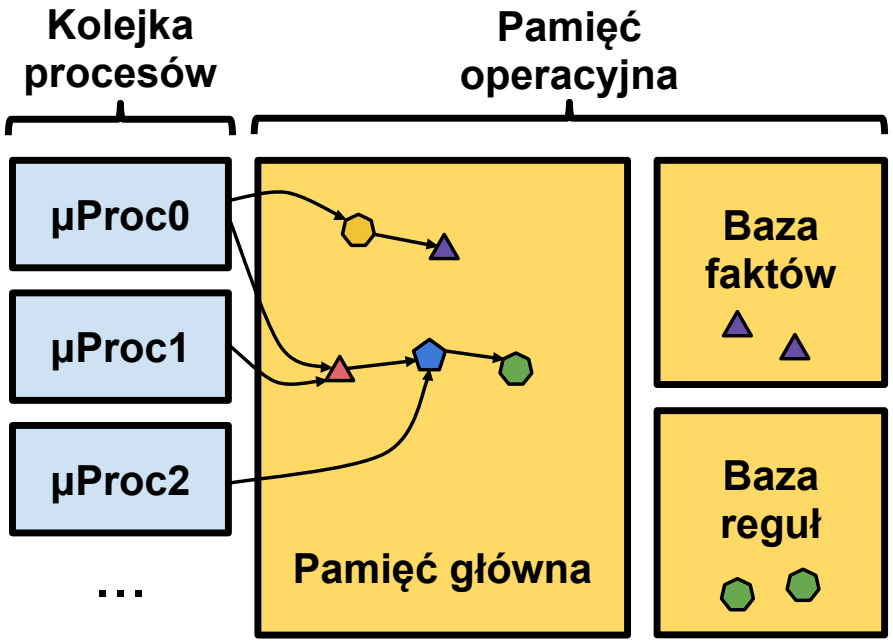
– add a code example contrasting closure conversion and lambda lifting

4. Środowisko uruchomieniowe języka

– briefly touch on the architecture

– mention Scheme bootstrap

4.1. Architektura środowiska uruchomieniowego



Rysunek 4.1: Schemat architektury środowiska uruchomieniowego języka FOOF.

– describe various parts

– mention that this is single threaded and requires forking for real concurrency

– hint at in-depth description of RBS implementation in a future section

4.2. Implementacja podstawowych typów danych

– describe scheme bootstrap [[7]]

– describe equivalence of various constructs such as lambdas

4.3. Implementacja kontynuacji

– describe how continuations are handled without getting into CFS (returning cont + hole aka trampoline, contrast to how G-machine/TIM reductions work) [[17]] [[12]]

– add a code example with step-by-step execution

– hint at debugging potential using step by step continuation execution with debug info inbetween

4.4. Implementacja obsługi wyjątków

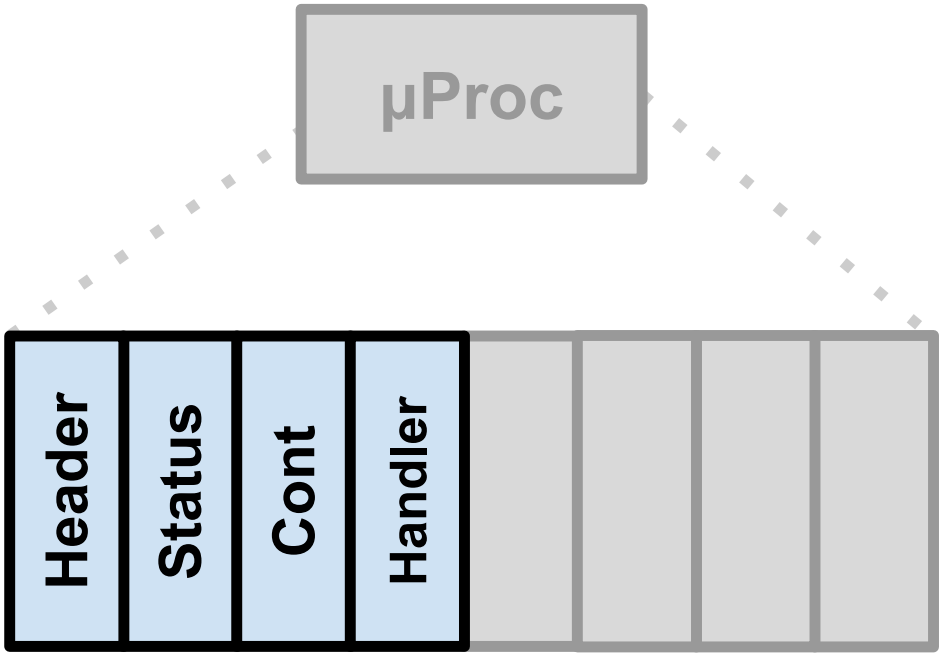
– describe how continuations are used for error handling - handle & raise

– note about restarts

– note about implementing letcc using handle & raise ?

4.5. Implementacja procesów

– add a diagram of the uProc context - only include status, cont & handler registers



Rysunek 4.2: Schemat kontekstu procesu obrazujący rejestry niezbędne do jego działania.

– describe uProc context registers

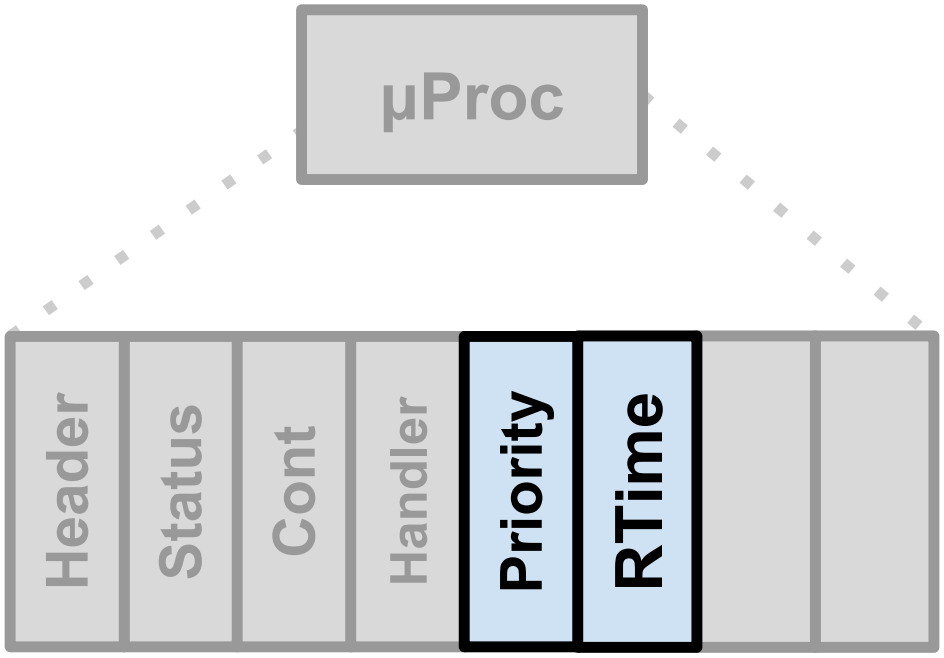
– describe how trampolines play into this scheme (recall `&yield-cont`)

– contrast trampolines with corutines (more suitable in CPS) and yielding (done implicitly) [[42]]

– describe how error handling is implemented (recall `&uproc-error-handler` etc)

– contrast with erlang [[22]]

4.6. Harmonogramowanie procesów

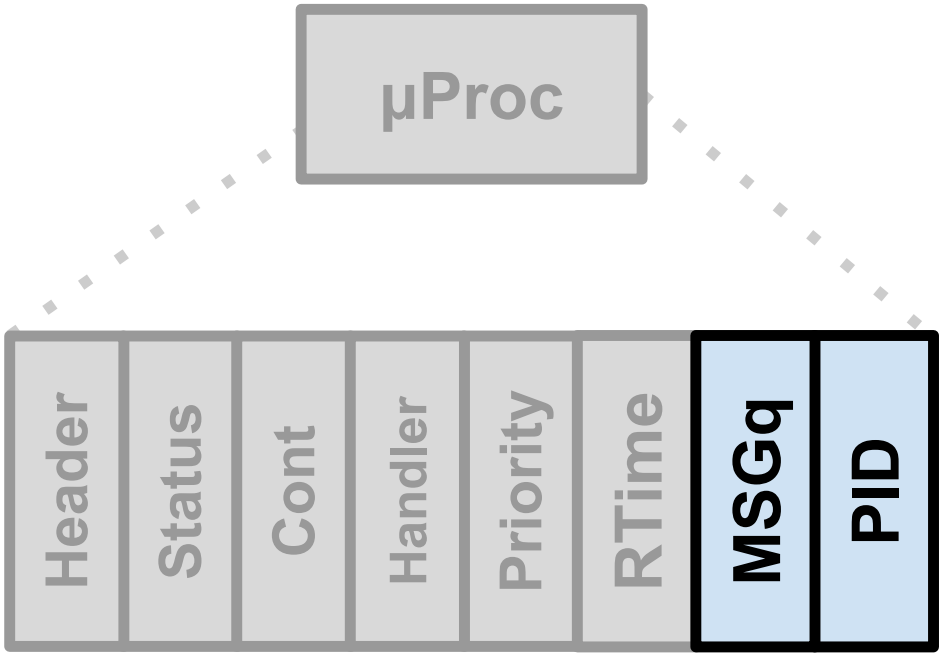


Rysunek 4.3: Dodatkowe rejestry kontekstu mikroprocesu wymagade do implementacji algorytmu *Completely Fair Scheduler*.

- describe the Completely Fair Scheduler [[43]]
- add pseudocode listing showing the algorithm
- describe uProc context switching
- contrast current impl with previous one (lack of wait list - notifications, heaps instead of RBT, number of reductions instead of time) [[44]]
- contrast with erlang [[22]]

4.7. Implementacja Modelu Aktorowego

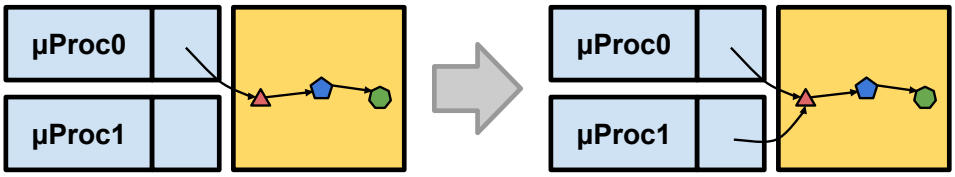
- describe actor model briefly [[20]] [[21]]



Rysunek 4.4: Dodatkowe rejestry kontekstu mikroprocesu wymagade do implementacji Modelu Aktorowego.

– describe modifications to the runtime required by actor model (**current-uproc**, uproc list, context fields)

– describe implementation of various actor model primitives



Rysunek 4.5: Diagram obrazujący efekty przekazywania wiadomości pomiędzy mikroprocesami.

– add some code examples and discussion of its effects and what happens

– contrast with erlang [[22]]

4.8. Dystrybucja obliczeń

– difference between concurrency & distribution

– describe modifications to the runtime in order to support distribution

– hint about using a simple protocol

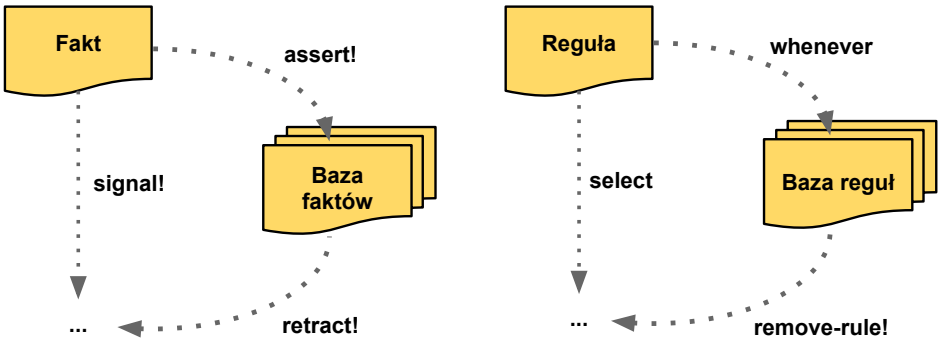
– hint about moving this into stdlib

5. Reprezentacja i przetwarzanie wiedzy

– describe how this needs a separate section

– elaborate on different ways of knowledge representation [[25]] [[45]] [[23]] [[?]] [[?]]

5.1. Reprezentacja wiedzy w języku



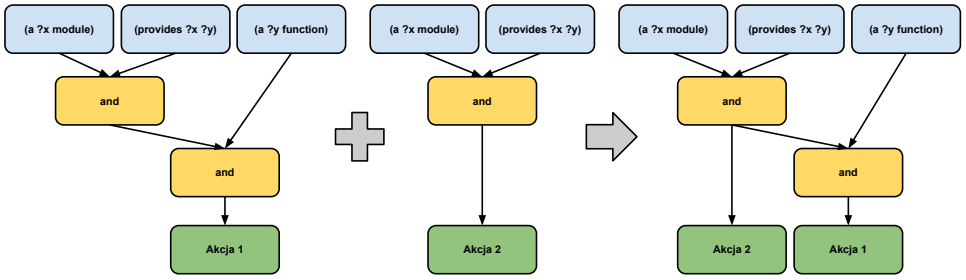
Rysunek 5.1: Schemat działania wbudowanych baz faktów i reguł.

– describe facts - signalling, assertion & retraction

– describe rules briefly - adding & disabling, triggering

5.2. Algorytm Rete

– describe in detail the algorithm [[46]]



Rysunek 5.2: Schemat łączenia podsieci w algorytmie *Rete*.

– describe briefly its history [[47]]

– Rete vs naïve approach (vs CLIPS or similar ?)

– add a benchmark diagram showing how Rete is better

– contrast it with other algorithms [[48]]

5.3. Implementacja Rete - wnioskowanie w przód

– describe what forward-chaining is

– describe naïve Rete - no network merging

– hint that this might be a good thing (future section)

– describe all the nodes [[46]]

5.4. Implementacja wnioskowania wstecz

– describe what backward-chaining is

– describe fact store in detail - linear, in-memory database

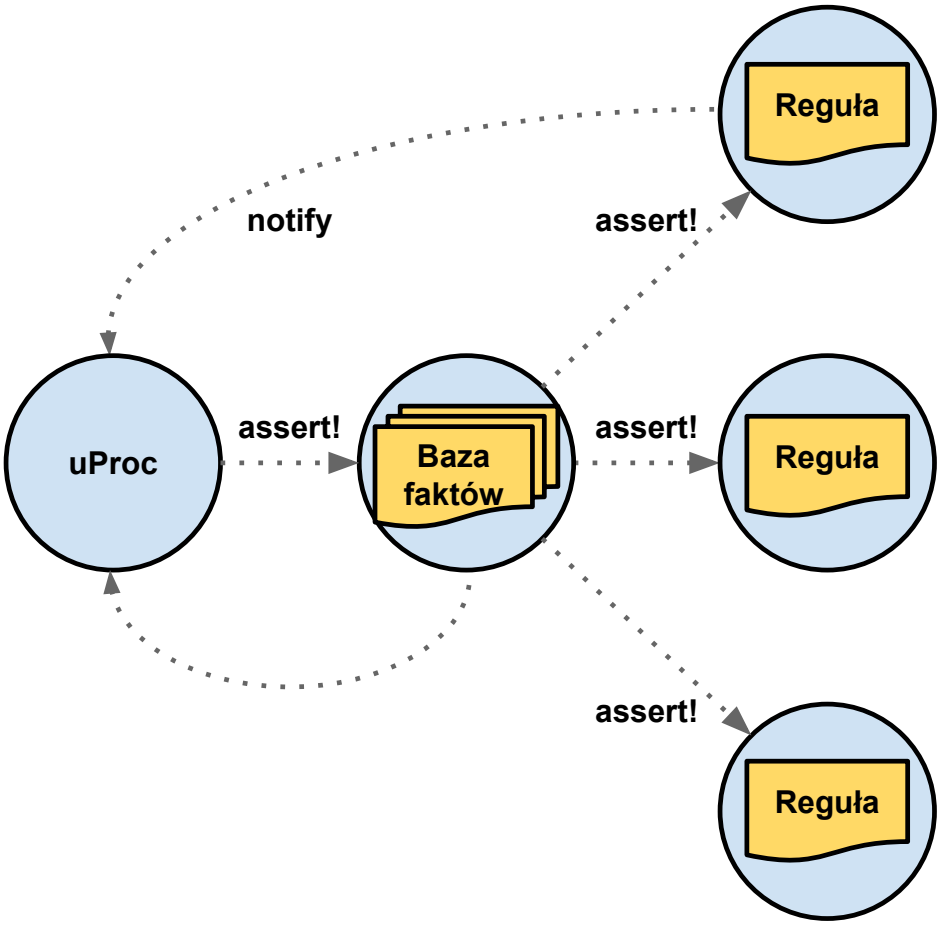
– querying fact store = create a rule and apply all known facts to it

5.5. Integracja z Systemem Uruchomieniowym

– describe how it sucks right now (notify-whenever instead of generic whenever, logic rule removal)

– describe possible integration with the module system (fact inference)

– describe possible representation of rules by autonomus processes [[49]]



Rysunek 5.3: Schemat działania rozproszonej wersji algorytmu *Rete*.

– hint at movig the implementation to the stdlib

6. Podsumowanie

– reiterate the goal of the thesis

– state how well has it been achieved

6.1. Kompilator języka FOOF

– needs better optimizations

– needs better error handling

6.2. Środowisko uruchomieniowe

– needs more stuff

– needs macroexpansion

– needs to drop RBS and move it into stdlib

6.3. Przyszłe kierunki rozwoju

– more datatypes

– native compilation via LLVM

– bootstrapping compiler

– librarized RBS

– librarized distribution with data encryption & ACLs

– data-level paralellism

Bibliografia

[1] P. E. McKenney, “Is parallel programming hard, and, if so, what can you do about it?.” Free online version.

[2] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[3] J. Backus, “Can programming be liberated from the von neumann style?: A functional style and its algebra of programs,” *Commun. ACM*, vol. 21, pp. 613–641, Aug. 1978.

[4] J. Höller, V. Tsiatsis, C. Mulligan, S. Karnouskos, S. Avesand, and D. Boyle, *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. Elsevier, Apr. 2014.

[5] M. Richards, *Software Architecture Patterns*. O’Reilly, Feb. 2015.

[6] C. A. R. Hoare, “Hints on programming language design.,” tech. rep., Stanford, CA, USA, 1973.

[7] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews, *Revised [6] Report on the Algorithmic Language Scheme*. New York, NY, USA: Cambridge University Press, 1st ed., 2010.

[8] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.

[9] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, part I,” *Commun. ACM*, vol. 3, pp. 184–195, Apr. 1960.

[10] A. Church, “A set of postulates for the foundation of logic part I,” *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366, 1932. http://www.jstor.org/stable/1968702Electronic Edition.

[11] A. Church, “A set of postulates for the foundation of logic part II,” *Annals of Mathematics*, vol. 34, no. 2, pp. 839–864, 1933.

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

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

[14] M. Felleisen and D. P. Friedman, "(Y Y) works!," 1991.

[15] K. Rzepecki, "Design of a programming language with support for distributed computing on heterogenous platforms." 2015.

[16] J. C. Reynolds, "The discoveries of continuations," *Lisp Symb. Comput.*, vol. 6, pp. 233–248, Nov. 1993.

[17] A. W. Appel, *Compiling with Continuations*. Cambridge University Press, 1992.

[18] R. Harper, "Programming in standard ml," 1998.

[19] R. K. Dybvig, S. P. Jones, and A. Sabry, "A monadic framework for delimited continuations," tech. rep., IN PROC, 2005.

[20] 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.

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

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

[23] S. Hachem, T. Teixeira, and V. Issarny, "Ontologies for the Internet of Things," in *Proceedings of the 8th Middleware Doctoral Symposium*, MDS '11, (New York, NY, USA), pp. 3:1–3:6, ACM, 2011.

[24] H. Samimi, C. Deaton, Y. Ohshima, A. Warth, and T. Millstein, "Call by meaning," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, (New York, NY, USA), pp. 11–28, ACM, 2014.

[25] D. S. C. G. Wang, W and K. Moessner, "Knowledge representation in the Internet of Things: Semantic modelling and its application," *Wang, W, De, S, Cassar, G and Moessner, K*, vol. 54, pp. 388 – 400.

[26] A. Bawden, "Quasiquotation in lisp," tech. rep., University of Aarhus, 1999.

[27] C. Queinnec, "Macroexpansion reflective tower," in *Proceedings of the Reflection'96 Conference*, pp. 93–104, 1996.

[28] A. Bawden, "First-class macros have types," in *In 27th ACM Symposium on Principles of Programming Languages (POPL'00*, pp. 133–141, ACM, 2000.

[29] J. N. Shutt, *Fexprs as the basis of Lisp function application or $vau: the ultimate abstraction*. PhD thesis, Worcester Polytechnic Institute, August 2010.

[30] J. M. Gasbichler, *Fully-parameterized, first-class modules with hygienic macros.* PhD thesis, Eberhard Karls University of Tübingen, 2006. http://d-nb.info/980855152.

[31] A. Rossberg, "1ML - core and modules united," 2015.

[32] A. Ghuloum, "An Incremental Approach to Compiler Construction," in *Scheme and Functional Programming 2006*.

[33] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[34] G. Hutton and E. Meijer, "Monadic parser combinators," 1996.

[35] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, (New York, NY, USA), pp. 111–122, ACM, 2004.

[36] A. Kennedy, "Compiling with continuations, continued," in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, (New York, NY, USA), pp. 177–190, ACM, 2007.

[37] D.-A. German, "Recursion without circularity or using let to define letrec," 1995.

[38] O. Kaser, C. R. Ramakrishnan, and S. Pawagi, "On the conversion of indirect to direct recursion," vol. 2, pp. 151–164, Mar. 1993.

[39] D. Bacon, "A Hacker's Introduction to Partial Evaluation," 2002.

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

[41] 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.

[42] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Trans. Program. Lang. Syst.*, vol. 31, pp. 6:1–6:31, Feb. 2009.

[43] C. S. Pabla, "Completely fair scheduler," *Linux J.*, vol. 2009, Aug. 2009.

[44] R. Sedgewick, "Left-leaning red-black trees," 2008.

[45] P. Barnaghi, W. Wang, C. Henson, and K. Taylor, "Semantics for the Internet of Things: Early progress and back to the future," *Int. J. Semant. Web Inf. Syst.*, vol. 8, pp. 1–21, Jan. 2012.

[46] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17 – 37, 1982.

[47] C. L. Forgy, *On the Efficient Implementation of Production Systems*. PhD thesis, Pittsburgh, PA, USA, 1979. AAI7919143.

[48] D. P. Miranker, *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. PhD thesis, New York, NY, USA, 1987. UMI Order No. GAX87-10209.

[49] A. Gupta, C. Forgy, A. Newell, and R. Wedig, "Parallel algorithms and architectures for rule-based systems," *SIGARCH Comput. Archit. News*, vol. 14, pp. 28–37, May 1986.

A. Gramatyka języka FOOF

– concrete language grammar in PEG or BNF

B. Przykładowe programy

Poniżej zaprezentowano przykładowe programy w języku `FOOF` i krótki opis ich działania. Programy mogą zostać skompilowane i uruchomione za pomocą udostępnionego interfejsu kompilatora i środowiska uruchomieniowego języka. W konsoli systemu należy w tym celu wywołać odpowiednio funkcje `compile` i `run` podając interesujący program jako parametr, na przykład:

```
> (compile 'program)
> (run 'program)
```

B.1. Hello world!

Program definuje funkcję `hello` obrazującą podstawowe operacje języka i następnie wywołuje ją z jednym parametrem. Po uruchomieniu program powoduje wypisanie wiadomości `Hello world!` na ekranie komputera.

```
(define (hello world)
  (if (= nil world)
      (raise 'nope)
      (do (display "Hello ")
          (display world)
          (display "!")
          (newline))))

(hello "world")
```

Listing 10: Popularny program *Hello world!*.

B.2. Funkcja Fibonacciego

Program prezentuje definicję funkcji Fibonacciego z wykorzystaniem konstrukcji `letrec`, służącej do definiowania funkcji rekursywnych. Następnie program oblicza wynik funkcji Fibonacciego dla liczby 23.

```
(letrec ((fib (lambda (n)
                (if (< n 2)
                    n
                    (+ (fib (- n 1))
                       (fib (- n 2)))))))
  (fib 23))
```

Listing 11: Definicja funkcji Fibonacciego.

B.3. Obsługa błędów

Program prezentuje wykorzystanie wbudowanego w język systemu obsługi błędów. Deklarowana jest procedura obsługi błędów, która restartuje obliczenia z nową wartością. Następnie program dwukrotnie sygnalizuje wystąpienie błędu. Wynikiem działania programu jest liczba 24.

```
(* 2 (handle (raise (raise 3))
           (lambda (e restart)
             (restart (* 2 e)))))
```

Listing 12: Zastosowanie wbudowanego mechanizmu obsługi błędów.

B.4. Model Aktorowy

Program korzysta z dwóch komunikujących się procesów do zobrazowania sposobu wykorzystania zaimplementowanego w języku Modelu Aktorowego. Efektem działania programu jest wypisanie wiadomości `Hello world!` na ekranie komputera.

```
(let ((pid (spawn (lambda ()
                    (let ((msg (recv)))
                      (display (cdr msg))
                      (newline)
                      (send (car msg) " world!"))))))
  (send pid (cons (self) "Hello"))
  (display (recv))
  (newline))
```

Listing 13: Wykorzystanie prymitywnych operacji Modelu Aktorowego.

B.5. Współbieżne obliczenia funkcji Fibonacciego

Program definiuje funkcję Fibonacciego oraz dodatkową funkcję wyświetlającą informacje o systemie. Następnie tworzone są trzy procesy współbieżnie obliczające wartość funkcji Fibonacciego dla liczby 30. Program periodycznie wyświetla różne informacje o działających procesach.

```
(letrec ((fib (lambda (n)
                (if (< n 2)
                    n
                    (+ (fib (- n 1))
                       (fib (- n 2))))))
         (monitor (lambda ()
                    (task-info)
                    (sleep 2000)
                    (monitor))))
  (spawn (lambda ()
           (fib 30)))
  (spawn (lambda ()
           (fib 30)))
  (spawn (lambda ()
           (fib 30)))
  (monitor))
```

Listing 14: Równoległe obliczanie funkcji Fibonacciego.

B.6. System modułowy

Program definiuje dwa moduły - `logger` oraz `test`. Moduł `test` wymaga do działania implementacji modułu logowania. Program tworzy instancję modułu `logger` i następnie tworzy instancję modułu `test` wykorzystując uprzednio zdefiniowany moduł logowania. Efektem działania programu jest wypisanie dwóch wiadomości na ekranie komputera. Wiadomości są odpowiednio sformatowane przez moduł `logger`.

```
(module (logger)
  (define (log level string)
    (display "[")
    (display level)
    (display "] ")
    (display string)
    (newline))

  (define (debug string)
    (log 'DEBUG string))

  (define (info string)
    (log 'INFO string))

  (define (warn string)
    (log 'WARN string))

  (define (error string)
    (log 'ERROR string)))

(module (test logger)
  (define (do-something)
    (logger.info "doing something")
    (logger.error "failed badly!")))

(let ((t (test (logger))))
  (t.do-something))
```

Listing 15: Wykorzystanie wbudowanego systemu modułowego.

B.7. Wnioskowanie w przód

Program prezentuje wykorzystanie wbudowanego w język systemu regułowego. Definiowane są trzy funkcje, jedna z nich co pewien czas sygnalizuje zajście pewnego zdarzenia - upływ czasu. Druga funkcja oczekuje notyfikacji od systemu regułowego i wyświetla informacje o przechwyconych zdarzeniach. Trzecia funkcja, jest pomocniczą funkcją wyświetlającą informacje o procesach uruchomionych w systemie. Następnie program definiuje prostą regułę i uruchamia wszystkie niezbędne procesy.

```
(letrec ((monitor (lambda ()
                    (task-info)
                    (sleep 10000)
                    (monitor)))
         (timer (lambda (t)
                  (signal! `(curr-time ,t))
                  (sleep 1000)
                  (timer (+ t 1))))
         (listen (lambda ()
                   (let ((t (recv)))
                     (display "Current time: ")
                     (display (cdr (car t)))
                     (newline)
                     (listen)))))
  (spawn (lambda () (timer 0)))
  (notify-whenever (spawn (lambda ()
                            (listen)))
                   '(curr-time ?t))
  (monitor))
```

Listing 16: Wykorzystanie wbudowanego systemu regułowego.

B.8. Obsługa złożonych zdarzeń

Program działa podobnie do przykładu z listingu 16. Definiowana jest złożona reguła, która notyfikuje proces nasłuchujący jedynie, gdy wartości powiązane z faktami `foo` oraz `bar` osiągają odpowiednie wartości.

```
(letrec ((monitor (lambda ()
                    (task-info)
                    (sleep 10000)
                    (monitor)))
         (notify (lambda (prefix t)
                   (assert! `(notify ,prefix ,(random)))
                   (sleep t)
                   (notify prefix t)))
         (listen (lambda ()
                   (let ((m (recv)))
                     (display "Complex event: ")
                     (display m)
                     (newline)
                     (listen)))))
  (notify-whenever (spawn listen)
                   '(filter (and (?notify foo ?foo)
                                 (?notify bar ?bar))
                            (>= ?foo 0.5)
                            (< ?foo 0.75)
                            (<= ?bar 0.1)))
  (spawn (lambda ()
           (notify 'foo 1000)))
  (spawn (lambda ()
           (notify 'bar 5000)))
  (monitor))
```

Listing 17: Zastosowanie wbudowanego systemu regułowego do obsługi złożonych zdarzeń.

B.9. Wnioskowanie wstecz

Program prezentuje wykorzystanie wnioskowania wstecz wbudowanego w język systemu regułowego. Na bazie faktów wykonywany jest szereg operacji, a następnie program odpytuje

bazę faktów o wartości, dla których wystąpiły fakty `foo` oraz `bar`. Wynikiem działania programu jest asocjacja `(?value . 2)`.

```
(assert! '(foo 1))
(assert! '(foo 2))
(assert! '(foo 3))
(assert! '(bar 2))
(assert! '(bar 3))
(retract! '(foo 2))
(signal! '(foo 4))

(select '(and (foo ?value)
              (bar ?value)))
```

Listing 18: Wykorzystanie wnioskowania wstecz.

C. Spis wbudowanych funkcji języka FOOF

- list contents of bootstrap.scm
- describe what `&make-structure`, `&yield-cont` etc do

D. Spisy rysunków i fragmentów kodu

Spis rysunków

Spis listingów