Spis treści

1 Wstęp

W referacie tym omówiona zostanie implementacja reduktora grafów oparta na *template instantation*. Jest to najprostsza możliwa implementacja języka funkcyjnego. Przykładowym językiem, którego dotyczy referat będzie język *Core* – minimalny język funkcyjny, do którego jednakże można sprowadzić języki funkcyjne wyższego poziomu.

2 Język *Core*

Referat ten dotyczy implementacji interpretera języka *Core*. Język ten jest minimalnym językiem funkcyjnym, jednak możliwa jest translacja programu w np. *Haskellu* do tego języka.

Na początek przykład programu w *Core*:

```
main = razyDwa 21
razyDwa x = x + x
```

Program w języku *Core* składa się ze zbioru *definicji superkombinatorów*. Superkombinator main odpowiada za początek programu. W tym przypadku, program obliczy 21 + 21.

2.1 Core *w* Haskellu

Poniżej znajduje się Haskellowy typ do reprezentacji wyrażenia w Core:

```
> module Language where
> import Utils
> data Expr a
>     = EVar Name             -- zmienne
>     | ENum Int              -- liczby
>     | EConstr Int Int       -- Constructor tag arity
```

```
>     | EAp (Expr a) (Expr a) -- aplikacje
>     | ELet                  -- wyrażenia let(rec)
>           IsRec             -- true, jeśli jest rekurencyjne
>           [(a, Expr a)]     –- definicje
>           (Expr a)          -- ciało let(rec)
>     | ECase                 -- wyrażenia ‘case’
>           (Expr a)          -- analizowane wyrażenie
>           [Alter a]         -- alternatywy
>     | ELam [a] (Expr a)     -- lambda
> type CoreExpr = Expr Name
```

Na przykład wyrażenie:

```
x + y
```

jest reprezentowane następująco:

```
EAp (EAp (EVar "+") (EVar "x")) (Evar "y")
```

Konstruktor `EVar` jest używany do reprezentowania zmiennych. Nazwa zmiennej jest opisana łańcuchem znaków, konstruujemy w tym celu synonim typu:

```
> type Name = String
```

Wyrażenia `let` oraz `letrec` (let zawierający odwołanie rekurencyjne - w Haskellu jest tylko jedno słowo kluczowe) są reprezentowane przez konstruktor `ELet` zawierający: flagę `IsRec` służącą do odróżniania `let` od `letrec`, listę definicji oraz wyrażenie będące ciałem `let(rec)`. Typ `IsRec` definiujemy następująco:

```
> type IsRec = Bool
> recursive, nonRecursive :: IsRec
> recursive = True
> nonRecursive = False
```

Każda definicja jest parą zmiennej i wyrażenia, do którego zmienna ta jest ograniczana. Definiujemy dwie użyteczne funkcje: `bindersOf` zwraca listę zmiennych oraz `rhssOf`, która zwraca listę prawych stron ograniczeń:

```
> bindersOf::[(a,b)]->[a]
> bindersOf defns = [name | (name, rhs)<-defns]
> rhssOf::[(a,b)]->[b]
> rhssOf defns =[rhs | (name, rhs)<-defns]
```

Wyrażenie `case` składa się z analizowanego wyrażenia oraz listy alternatyw. Każda alternatywa zawiera znacznik, listę ograniczanych zmiennych oraz wyrażenie na prawo od strzałki.

```
> type Alter a = (Int, [a], Expr a)
> type CoreAlt = Alter Name
```

Definiujemy jeszcze funkcję sprawdzającą, czy wyrażenie jest atomem, czyli czy nie zawiera wewnętrznej struktury.

```
> isAtomicExpr::Expr a->Bool
> isAtomicExpr (EVar v) = True
> isAtomicExpr (ENum n) = True
> isAtomicExpr _ = False
```

Ostatecznie program w języku Core jest po prostu listą definicji superkombinatorów.

```
> type Program a = [ScDefn a]
> type CoreProgram = Program Name
```

Definicja superkombinatora składa się z jego nazwy, listy argumentów (która może być pusta) oraz ciała.

```
> type ScDefn a = (Name, [a], Expr a)
> type CoreScDefn = ScDefn Name
```

Przykład prostego programu:

```
main = double 21
double = x + x
```

Reprezentacja w Haskellu:

```
[("main", [], (EAp (EVar "double") (ENum "21")),
 ("double", ["x"], (EAp (EAp (EVar "+") (EVar "x")) (EVar "x")))]
```

2.2 Standardowe preludium

Preludium dla Haskella (plik Prelude.hs) zawiera definicje podstawowych funkcji. Dla języka Core także definiujemy preludium, jednak będzie ono znacznie uboższe.

```
I x = x;
K x y = x;
K1 x y = y;
S f g x = f x (g x);
compose f g x = f (g x);
twice f = compose f f
```

Poniższa definicja określa program odpowiadający za preludium:

```
> preludeDefs :: CoreProgram
> preludeDefs
>  = [("I", ["x"], EVar "x",
>     ("K", ["x","y"], EVar "x"),
>     ("K1", ["x", "y"], EVar "y"),
>     ("S", ["f", "g", "x"], EAp (EAp (EVar "f") (EVar "x"))
>                                (EAp (EVar "g") (EVar "x"))),
>     ("compose", ["f", "g", "x"], EAp (EVar "f")
>                                      (EAp (EVar "g") (EVar "x"))),
>     ("twice", ["f"], EAp (EAp (EVar "compose") (EVar "f")) (EVar "f"))]
```

3 Program w *Core* w postaci grafu

3.1 Omówienie metody

- Program funkcyjny jest „wykonywany” poprzez *obliczenie wyrażenia*.
- Wyrażenie jest reprezentowane przez *graf*.
- Obliczenie następuje przez przeprowadzenie sekwencji *redukcji*.
- Redukcja zastępuje lub uaktualnia *wyrażenie redukowalne* przez jego formę zredukowaną. Sformułowaniu „wyrażenie redukowalne” odpowiada skrót *redex* – z ang. *reducible expression*.
- Obliczone wyrażenie jest w *postaci normalnej* – nie ma więcej redeksów.

W każdym momencie obliczania wyrażenia może być wybór pomiędzy różnymi redeksami, jednak niezależnie od tego, jaką kolejność redukowania wybierzemy, zawsze otrzymamy tę samą odpowiedź, tzn. postać normalną. Niestety, niektóre wybory kolejności mogą spowodować zawieszenie się obliczeń.

Jednakże jeśli jakakolwiek kolejność redukowania spowoduje zatrzymanie się obliczeń, to polityka wybierania zawsze najbardziej zewnętrznego redeksa także.

3.2 Przykład

Program w Core jest zbiorem definicji nazywanych *superkombinatorami*. Wykonanie programu rozpoczyna się od superkombinatora `main`. Rozważmy następujący banalny przykład:

```
kwadrat x = x * x
main = kwadrat (kwadrat 3)
```

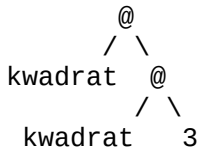
Na początku ewaluowane wyrażenie jest reprezentowane przez trywialny graf (a właściwie drzewo):

```
main
```

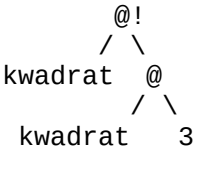
Ponieważ `main` sam w sobie jest redeksem – nie ma argumentów, zastępujemy je jego ciałem:

```
main
```

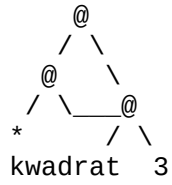
redukuje się do



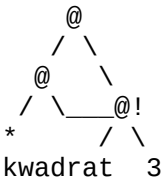
Aplikacje będą reprezentowane prze znak ‘@’. W tym momencie najbardziej zewnętrznym redeksem jest kwadrat. Żeby zredukować aplikację funkcji zamieniamy redex instancją ciała funkcji podstawiając wskaźnik do argumentu w miejscu każdego wystąpienia parametru formalnego. Korzeń nadpisywanego redeksa jest oznaczony przez ‘`!`’.



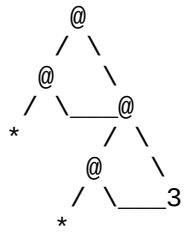
redukuje się do



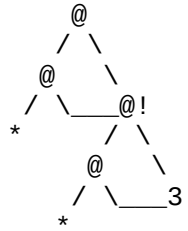
Jedyny pozostały redex, to wewnętrzna aplikacja `kwadrat 3`. Aplikacja * nie jest redukowalna, ponieważ * wymaga, aby jego argumenty zostały obliczone.



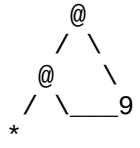
redukuje się do



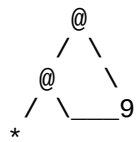
Pozostaje jeden redex, wewnętrzne mnożenie, które zastępujemy przez wynik.



redukuje się do



Uaktualnienie korzenia redeksa powoduje, że oba argumenty zewnętrznego mnożenia widzą rezultat mnożenia wewnętrznego. Pozostaje ostatni krok:



redukuje się do

```
81
```

3.3 Trzy kroki redukcji

Jak można było zobaczyć w poprzednim przykładzie, redukcja przebiega w trzech krokach, powtarzanych aż do uzyskania postaci normalnej:

1. Znajdujemy następny redex.
2. Redukujemy go.
3. Uaktualniamy korzeń redeksa przez wynik.

Są dwa rodzaje redeksów, każdy redukowany w inny sposób.

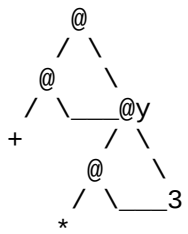
Redukcja superkombinatorów.

Jeśli najbardziej zewnętrzna aplikacja funkcji jest aplikacją superkombinatora, to z pewnością jest redeksem. Redukcja superkombinatorów polega na podstawieniu argumentów do jego ciała. Ciało superkombinatora może zawierać wyrażenia `let` lub `letrec`. Na przykład:

```
f x = let y = x*x
      in y+y
```

Wyrażenia `let` i `letrec` są traktowane jako *dosłowne opisy grafu*. Przykład użycia `f`:

redukuje się do



Wyrażenie `let` definiuje podwyrażenie `x*x` nazwane `y`. Ciało `let`, czyli `y+y`, używa wskaźników do podwyrażenia w miejscu `y`. Zatem zwykłe wyrażenia opisują drzewa, wyrażenia `let` – grafy acykliczne, a `letrec` – grafy cykliczne.

Redukcja wbudowanych funkcji pierwotnych.

Jeśli najbardziej zewnętrzna aplikacja funkcji jest aplikacją funkcji pierwotnej, to nie musi być redeksem. Zależy to od tego, czy jej argumenty zostały już obliczone. Jeśli nie, to przechodzimy do redukcji jej argumentów, która odbywa się w ten sam sposób, jak redukcja całego wyrażenia. Po obliczeniu argumentów wracamy do redukcji zewnętrznej aplikacji.

Pierwszym krokiem cyklu redukcji jest miejsca następnej redukcji. Łatwo jest znaleźć najbardziej zewnętrzną aplikację funkcji (jednak nie koniecznie musi być ona redukowalna):

1. Zaczynając od korzenia, podążaj lewą gałęzią węzłów aplikacji, aż napotkasz superkombinator albo funkcją pierwotną. Lewa gałąź węzłów aplikacji jest nazywana *kręgosłupem* wyrażenia, a proces ten *rozwijaniem* kręgosłupa. Zwykle korzysta się ze stosu w celu zapamiętania adresów węzłów napotkanych po drodze.
2. Teraz sprawdzamy ile argumentów superkombinator lub funkcja pierwotna przyjmuje i cofamy się o tyle węzłów w gorę. Tym sposobem docieramy do korzenia najbardziej zewnętrznej aplikacji funkcji.

W ten sposób znajdujemy najbardziej zewnętrzną aplikację funkcji. Jednak nie musi być ona redeksem – może się zdarzyć, że jest funkcją pierwotną, której argumenty nie zostały obliczone. W takim przypadku musimy obliczyć potrzebne argumenty. Nasz stos zapamiętujemy i rozpoczynamy z nowym stosem redukcję argumentu w ten sam sposób.

Zamiast zapamiętywać cały stos, będziemy nowy budowali na starym, a pamiętać będziemy jedynie granicę między nowym a starym stosem.

4 System zmiany stanów

System zmiany stanów będzie potrzebny do opisu implementacji redukcji grafu. Jest to sposób zapisu opisujący zachowanie się maszyny sekwencyjnej. W każdym momencie maszyna jest w jakimś *stanie*, poczynając od wyróżnionego *stanu początkowego*. Jeżeli stan maszyny odpowiada jednej z *reguł zmiany stanu*, to maszyna zmienia swój stan na opisany przez daną regułę. Kiedy nie ma odpowiednich reguł, maszyna się zatrzymuje.

Przykładowa maszyna obliczająca iloczyn dwóch liczb. Maszyna jest opisana za pomocą dwóch reguł:

	n	*m*	*d*	*t*
→	*n*	*m*	*d -1*	*t + 1*
	gdzie *d > 0*			

	n	*m*	*0*	*t*
→	*n*	*m - 1*	*n*	*t*
	gdzie *m > 0*			

Każdy stan jest czwórką: *(n, m, d, t)*, mnożone liczby to *m* i *n*, wynik zwracany przez *t*. Stan początkowy to *(n, m, 0, 0)*.

System zmiany stanów możemy łatwo przenieść do Haskella. Na początku definiujemy synonim typu opisującego stan.

```
> type MultState = (Int, Int, Int, Int)
```

Funkcja evalState przyjmuje stan i zwraca listę wszystkich stanów następujących po nim.

```
> evalMult::MultState->[MultState]
> evalMult state =
>     if multFinal state
>           then [state]
>           else state : evalMult (stepMult state)
```

Funkcja stepMult przyjmuje stan, który nie jest końcowy i zwraca następny.

```
stepMult::MultState->MultState
stepMult (n, m, d, t) | d >  0 = (n, m,   d-1, t+1)
stepMult (n, m, d, t) | d == 0 = (n, m-1, n,   t)
```

Funkcja multFinal testuje, czy jej argument jest stanem końcowym.

```
> multFinal::MultState->Bool
> multFinal (_, 0, 0, _) = True
> multFinal _ = False
```

5 Implementacja reduktora grafów w Haskellu

5.1 Reguły zmiany stanów dla reduktora grafów

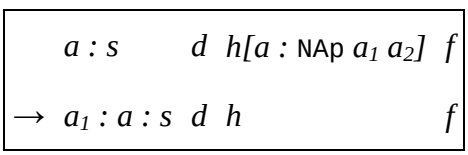
Stan maszyny redukującej graf jest opisany za pomocą czwórki *(stos, zrzut, sterta, globalne)*, w skrócie *(s, d, h, f)* -- z ang. (stack, dump, heap, globals).

- *Stos* jest stosem *adresów*, z których każdy identyfikuje *węzeł* na stercie. Węzły te tworzą kręgosłup obliczanego wyrażenia. Notacja $a_1 : s$ oznacza stos, którego górnym elementem jest a_1, a reszta to *s*.
- *Zrzut* przechowuje stan stosu kręgosłupa przed obliczaniem argumentu funkcji pierwotnej.
- *Sterta* jest kolekcją oznaczonych *węzłów*. Notacja *h[a : węzeł]* oznacza, że w stercie *h* adres *a* odnosi się do węzła *węzeł*.
- Dla każdego superkombinatora i każdej funkcji pierwotnej, *globalne* zwraca adres węzła sterty jemu odpowiadającej.

Węzeł sterty może przyjąć trzy postaci:

- `NAp` a_1 a_2 odpowiada aplikacji węzła, którego adres to a_1 do tego, którego adres to a_2.
- `NSupercomb` *arg ciało* reprezentuje superkombinator argumentach to *arg*, a ciele *ciało*.
- `NNum` *n* reprezentuje liczbę *n*.

Pierwsza reguła opisuje jak rozwinąć pojedynczy węzeł aplikacji na stosie kręgosłupa:



Element sterty w drugiej linijce tej reguły ciągle zawiera mapowanie adresu a do NAp a_1 a_2. Stosowanie tej reguły rozwinie cały kręgosłup wyrażenia na stosie, do momentu, gdy węzeł na szczycie stosu przestanie być węzłem typu NAp.

Druga reguła opisuje jak przeprowadzić redukcję superkombinatora.

$a_0 : a_1 : \ldots : a_n : s \;\; d \;\; h[a_0 :$ `NSupercomb` $[x_1, \ldots, x_n]$ *ciało*$] \;\; f$

$\rightarrow \qquad a_r : s \;\; d \;\; h' \qquad f$

gdzie $(h', \; a_r) =$ *instantiate ciało h* $f[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$

Najważniejsza w tej regule jest funkcja *instantiate* – podstaw. Jej argumenty to:

- wyrażenie, w którym wystąpienia zmiennych mają zostać zastąpione konkretnymi wartościami,
- sterta,
- globalne mapowanie nazw do adresów stert, powiększone o mapowanie nazw argumentów do ich adresów uzyskane ze stosu.

Zwraca nową stertę oraz adres korzenia stworzonej instancji. Taka skomplikowana instrukcja jest sprzeczna z duchem systemu zmiany stanów, gdzie każdy krok rozumiany jest jako prosta, atomowa operacja, ale taka jest natura *template instantation machine.*

Należy zauważyć, że reguła ta nie wpływa na korzeń redeksa; jest on zaledwie zastępowany na stosie przez korzeń wyniku.

5.2 Struktura implementacji

Nasza implementacja będzie miała następującą strukturę:

1. Tłumaczenie programu do postaci, którą będzie można wykonać. Odpowiedzialna za to będzie funkcja compile, która z programu produkuje początkowy stan maszyny *template instantation.*

```
> compile::CoreProgram->TiState
```

2. Wykonanie programu, przez aplikowanie kolejnych pasujących reguł, aż do momentu dotarcia do stanu końcowego. Wynik jest listą wszystkich stanów, przez które przeszła maszyna.

```
> eval::TiState->[TiState]
```

W tej implementacji pominięty zostanie parser oraz wyświetlanie wyników.

Program będzie wykonywany przez funkcję run:

```
> run::CoreProgram->[TiState]
> run = eval.compile
```

5.2.1 Kompilator

Typy danych

Musimy zdefiniować typ opisujący stan naszej maszyny:

```
> type TiState = (TiStack, TiDump, TiHeap, TiGlobals, TiStats)
```

Stan maszyny jest piątką, której pierwsze cztery elementy zostały opisane wcześniej (5.1), a piąty element służy do zbierania statystyk.

- *Stos kręgosłupa* jest po prostu stosem *adresów sterty*.

```
> type TiStack = [Addr]
```

Stos jest reprezentowany przez listę. Elementy stosu należą do abstrakcyjnego typu Addr, zdefiniowanego w module `utils` (`type Addr = Int`). Reprezentują adresy sterty.

- Sterta jest reprezentowana przez abstrakcyjny typ danych `Heap`, zdefiniowany w module `utils`. Musimy określić co zawiera sterta, to znaczy obiekty typu node:

```
> type TiHeap = Heap Node
> data Node
>        = NAp Addr Addr                        -- Aplikacja
>        | NSupercomb Name [Name] CoreExpr      -- Superkominator
>        | NNum Int                             -- Liczba
```

Powyższy typ odpowiada liście możliwości z 5.1, poza tym, że do superkombinatora zostało dodane pole typu `Name`, które przechowuje jego nazwę.

- Element *globalne* stowarzysza nazwę każdego superkombinatora z adresem węzła na stercie zawierającego jego definicję.

```
> type TiGlobals = ASSOC Name Addr
```

- Element `TiStats` jest używany jedynie do zbierania informacji o wydajności maszyny

```
> type TiStats = Int
```

Kompilator

Zadaniem kompilatora jest przerobienie programu na początkowy stan maszyny:

```
> compile program
>     = (initial_stack, initialTiDump, initial_heap, globals, tiStatInitial)
>     where
>     sc_defs = program ++ preludeDefs ++ extraPreludeDefs
>
>       (initial_heap, globals) = buildInitialHeap sc_defs
>
>     initial_stack = [address_of_main]
>     address_of_main =
>           aLookup globals "main" (error "main nie jest zdefiniowana")
```

`sc_defs` jest po prostu listą wszystkich zdefiniowanych superkombinatorów. `preludeDefs` jest zdefiniowane w module `Language`, a `extraPreludeDefs` jest listą standardowych funkcji, które, być może, będziemy chcieli mieć w przyszłości. Na razie jest to lista pusta:

```
> extraPreludeDefs = []
```

Następna definicja wykorzystuje pomocniczą funkcję, `buildInitialHeap`, do stworzenia początkowej sterty zawierającej węzeł `NSupercomb` dla każdego superkombinatora oraz listę globalnych odwzorowującej nazwę każdego superkombinatora na adres jego węzła.

Definicja `initial_stack` zawiera tylko jeden element – adres węzła superkombinatora main.

Definicja `buildInitialHeap` wykorzystuje funkcję `mapAccuml`.

Funkcja `mapAccuml` – przyjmuje ona trzy argumenty: funkcję *f*, akumulator acc oraz listę *[x1, …, xn]*. Funkcja ta wykonuje *f* dla każdego *xi* oraz akumulatora, *f* zwraca parę: element na liście wyników oraz nową wartość akumulatora. `mapAccuml` ostatecznie zwraca parę wyników: końcową wartość akumulatora acc' oraz listę poszczególnych wyników *f [y1, …, yn]*

W naszym przypadku akumulator jest stertą o początkowej wartości `hInitial` (pusta sterta). Lista *[x1, …, xn]* to lista definicji superkombinatorów `sc_defs`, wynikiem zaś jest lista asocjacji nazw superkombinatorów oraz adresów, `sc_addrs`.

```
> buildInitialHeap::[CoreScDefn] -> (TiHeap, TiGlobals)
> buildInitialHeap sc_defs = mapAccuml allocateSc hInitial sc_defs
>
> allocateSc::TiHeap->CoreScDefn->(TiHeap, (Name, Addr))
> allocateSc heap (name, args, body)
>     = (heap', (name, addr))
>     where
>     (heap', addr) = hAlloc heap (NSupercomb name args body)
```

Funkcja `allocateSc` alokuje pojedynczy superkombinator, zwracając nowy stos oraz element listy `sc_addrs`.

5.2.2 Ewaluacja

Funkcja `eval` zawsze zwraca aktualny stan jako pierwszy element wyniku. Jeżeli jest stan końcowy, nie są zwracane żadne inne stany. W przeciwnym wypadku następuje rekurencyjne wywołanie `eval` do następnego stanu. Jest to wykonywane przez pojedynczy krok (`step`), a następnie wywołanie `doAdmin` w celu przeprowadzenia czynności administracyjnych.

```
> eval state = state : rest_states
>     where
>     rest_states
>           | tiFinal state = []
>           | otherwise = eval next_state
>     next_state = doAdmin (step state)
>
> doAdmin :: TiState -> TiState
> doAdmin state = applyToStats tiStatIncSteps state
```

Sprawdzanie stanu końcowego

Ewaluacja została zakończona, jeśli na stosie pozostał pojedynczy obiekt, który jest albo liczbą, albo obiektem danych.

```
> tiFinal :: TiState -> Bool
> tiFinal ([sole_addr], dump, heap, globals, stats)
>     = isDataNode (hLookup heap sole_addr)

> tiFinal ([], dump, heap, globals, stats) = error "Pusty stos!"
> tiFinal state = False       -- Stos zawiera wiecej, niz jeden obiekt
```

Element na stosie jest zawsze adresem, więc musimy sprawdzić na stercie, czy jest to liczba czy nie.

Funkcja `isDataNode` sprawdza, czy obiekt to dane.

```
> isDataNode :: Node -> Bool
> isDataNode (NNum n) = True
> isDataNode node = False
```

Pojedynczy krok

Musimy przeprowadzić analizę typu elementu na szczycie stosu, więc najpier go z tamtąd „wyjmujemy”, a następnie korzystamy z funkcji pomocniczej `dispatch`.

```
> step :: TiState -> TiState
> step state
>     = dispatch (hLookup heap (hd stack))
>     where
```

```
>     (stack, dump, heap, globals, stats) = state
>
>     dispatch (NNum n) = numStep state n
>     dispatch (NAp a1 a2) = apStep state a1 a2
>     dispatch (NSupercomb sc args body) = scStep state sc args body
```

Błędem jest, jeśli na szczycie stosu znajduje się liczba – liczbyn nie powinny być traktowane jak funkcje.

```
> numStep :: TiState -> Int -> TiState
> numStep state n = error "Aplikacja liczby!"
```

Postępowanie z węzłem aplikacji jest opisane przez regułę rozwijającą, którą łatwo przenieść do Haskella.

```
> apStep :: TiState -> Addr -> Addr -> TiState
> apStep (stack, dump, heap, globals, stats) a1 a2
>     = (a1 : stack, dump, heap, globals, stats)
```

Aplikowanie superkombinatora

Żeby zaaplikować superkombinator, musimy wytworzyć egzemplarz jego ciała, łącząc nazwy argumentów z ich adresami ze stosu. Następnie wyrzucamy argumenty oraz korzeń redeksa ze stosu i włożyć na stos korzeń wyniku.

```
> scStep :: TiState -> Name -> [Name] -> CoreExpr -> TiState
> scStep (stack, dump, heap, globals, stats) sc_name arg_names body
>     = (new_stack, dump, new_heap, globals, stats)
>     where
>     new_stack = result_addr : (drop (length arg_names+1) stack)
>
>     (new_heap, result_addr) = instantiate body heap env
>     env = arg_bindings ++ globals
>     arg_bindings = zip2 arg_names (getArgs heap stack)
```

Żeby aplikować superkombinatory oraz funkcje pierwotne, potrzebna jest funkcja pomocnicza. Funkcja getArgs przyjmuje jako argument stos, na szczycie którego musi się znajdować superkombinator i zwraca listę stworzoną z argumentów każdego węzła aplikacji na stosie.

```
> getArgs :: TiHeap -> TiStack -> [Addr]
> getArgs heap (sc:stack)
>     = map get_arg stack
>     where get_arg addr = arg where (NAp fun arg) = hLookup heap addr
```

Funkcja instantiate pobiera wyrażenie, stertę oraz środowisko kojarzące nazwy z argumentami. Tworzy na stercie instancję wyrażenia i zwraca nową stertę orza adres do korzenia instancji. Środowisko jest używane przez instantiate, żeby określić adresy superkombinatorów oraz zmiennych lokalnych, które mają zostać zastąpione.

```
> instantiate :: CoreExpr     -- Cialo superkombinatora
>     -> TiHeap               -- Sterta przed instantacja
>     -> ASSOC Name Addr      -- Asocjacja nazw do adresów
>     -> (TiHeap, Addr)       -- Sterta po instantacji
>                             -- oraz adres korzenia instancji
```

Przypadek dla liczb:

```
> instantiate (ENum n) heap env = hAlloc heap (NNum n)
```

W przypadku aplikacji, tworzymy egzemplarz dwóch gałęzi oraz tworzymy węzeł aplikacji.

```
> instantiate (EAp e1 e2) heap env
>     = hAlloc heap2 (NAp a1 a2) where
>           (heap1, a1) = instantiate e1 heap env
>           (heap2, a2) = instantiate e2 heap1 env
```

W przypadku zmiennej szukamy po prostu jej nazwy w otrzyanym środowisku. Jeśli nie zostanie odnaleziona, to wyświetlany jest komunikat o błędzie:

```
> instantiate (EVar v) heap env
>     = (heap, aLookup env v (error ("Nie okrslona nazwa " ++ show v)))
```