Metody Realizacji Języków Programowania

Implementacja języków funkcyjnych

Marcin Benke

MIM UW

22 stycznia 2015

Specyficzne cechy języków funkcyjnych

Funkcje są pełnoprawnymi obywatelami:

1. Mogą być argumentami funkcji
2. Mogą być wynikami funkcji
3. Mogą być częściowo aplikowane
4. Mogą być tworzone anonimowo
5. Obliczenia mogą być leniwe, tzn. wartość argumentu jest wyliczana nie w momencie wywołania funkcji, ale w momencie jego (pierwszego) użycia.
6. Nie ma przypisania (tylko obliczanie wartości wyrażeń); w czystych językach funkcyjnych zasadniczo nie ma w ogóle efektów ubocznych

Przykład

Spójrzmy na prosty przykład ilustrujący te cechy

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x):(map f xs)
increaseAll :: [Int] -> [Int]
increaseAll = map (\x->x+1)
```

Funkcja `map` bierze funkcję $(a \to b)$ i $[a]$, dając w wyniku $[b]$.

Inne odczytanie: argumentem jest funkcja typu $(a \to b)$, zaś wynikiem. . . funkcja typu $[a] \to [b]$.

Funkcja `increaseAll` korzysta z tego drugiego odczytania, stosując funkcję `map` z jednym argumentem, którym jest anonimowa funkcja dodająca jeden do swego argumentu.

Wpływ na implementację

Ad 1 Funkcje jako argumenty funkcji występują w innych językach. Możemy jednak być zmuszeni do tworzenia i przekazywania domknięcia funkcji dla dostępu do zmiennych nielokalnych.

Ad 2 Jeśli funkcja może dawać w wyniku funkcję, dostęp do zmiennych nielokalnych jest problematyczny.

- Nie możemy skorzystać z klasycznego stosu rekordów aktywacji jak w językach imperatywnych.
- Domknięcie funkcji musi przechowywać wartości wszystkich zmiennych nielokalnych, z których ta funkcja korzysta.
- Możemy jednak przekształcić program do takiej (równoważnej) postaci, aby żadna funkcja nie korzystała ze zmiennych nielokalnych. Transformację tę przedyskutujemy nieco później.

Wpływ na implementację

3–5 W trakcie wykonania programu musimy przechowywać struktury (grafy) reprezentujące nieobliczone (lub częściowo tylko obliczone) wyrażenia.

- Wykonanie programu będzie polegało na konstruowaniu i redukowaniu takich grafów.
- Grafy, a nie drzewa zwn. wspólne podwyrażenia.
- W językach leniwych grafy nie muszą być acykliczne.
- Jak zobaczymy za chwilę, podejście takie jest wygodne także z innych powodów.

Ad 6 Brak efektów ubocznych umożliwia stosowanie na szeroką skalę transformacji programów do równoważnej, ale wygodniejszej lub efektywniejszej postaci.

Domknięcia

Problem zmiennych nielokalnych jest nam już znany.

Standardowym rozwiązaniem jest reprezentowanie wartości funkcyjnej przez tzw. domknięcie — wartość, z której można utworzyć wcielenie danej funkcji wszędzie, gdzie może to być potrzebne.

Co powinno zawierać takie domknięcie — zależy od konkretnego języka, a nawet od implementacji.

W Pascalu wystarczy gdy domknięciem jest para (adres funkcji, SL)
— ramka SL nie zniknie przedwcześnie ze stosu.

Kiedy funkcje mogą być wynikami funkcji, własność ta nie może być zagwarantowana.

Za domknięcie przyjmuje się wtedy zwykle informację o wartościach wszystkich zmiennych wolnych. Ich ilość (a zatem rozmiar domknięcia) można wyznaczyć statycznie.

Jeśli są efekty uboczne — adresy zmiennych zamiast ich wartości.

Superkombinatory

- Pewne funkcje możemy bezpiecznie przekazywać bez dodatkowych informacji.
- Są to te, które... nie mają zmiennych wolnych!
- Na tej obserwacji bazuje następująca metoda implementacji języków funkcyjnych: transformujemy program do postaci, w której żadna funkcja nie ma zmiennych wolnych.
- Funkcje takie nazywa się *superkombinatorami*, a proces transformacji *lambda-liftingiem*.

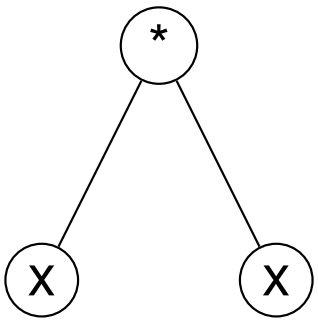
Na razie będziemy zakładać, że program dany jest w postaci ciągu definicji superkombinatorów. Domknięciami i lambda-liftingiem zajmiemy się w dalszej części wykładu.

Grafy wyrażeń

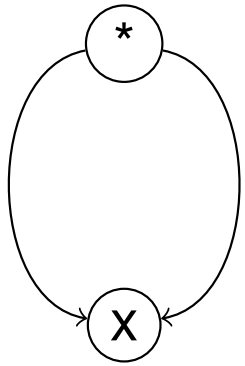
Graf wyrażenia jest uogólnieniem drzewa wyrażenia. Rozważmy na przykład funkcję:

```
kwadrat x = x * x
```

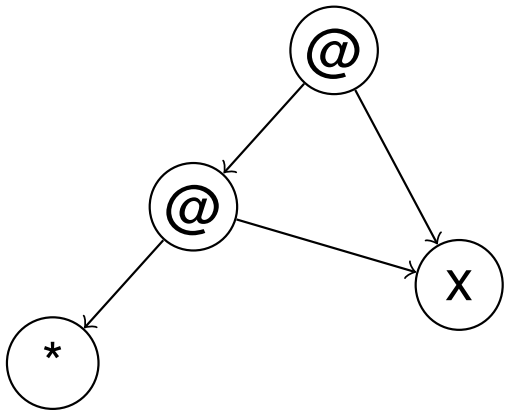
Możemy ją przedstawić w postaci drzewa



albo grafu



Ponieważ wyrażenie x*x oznacza zastosowanie funkcji * do argumentów x oraz x, dokładniejszą będzie reprezentacja:

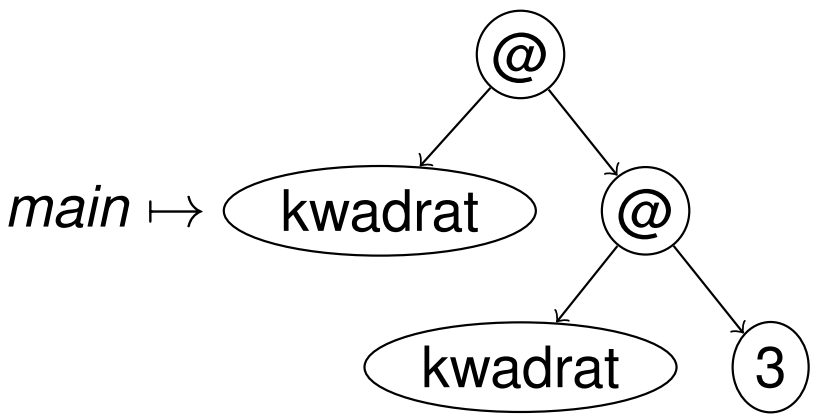


Redukcje grafów

Wykonanie programu funkcyjnego polega na obliczeniu wartości wyrażenia poprzez kolejne redukcje.
Przy reprezentacji grafowej redukcje te dokonywane są na grafach wyrażeń.
Przykład:

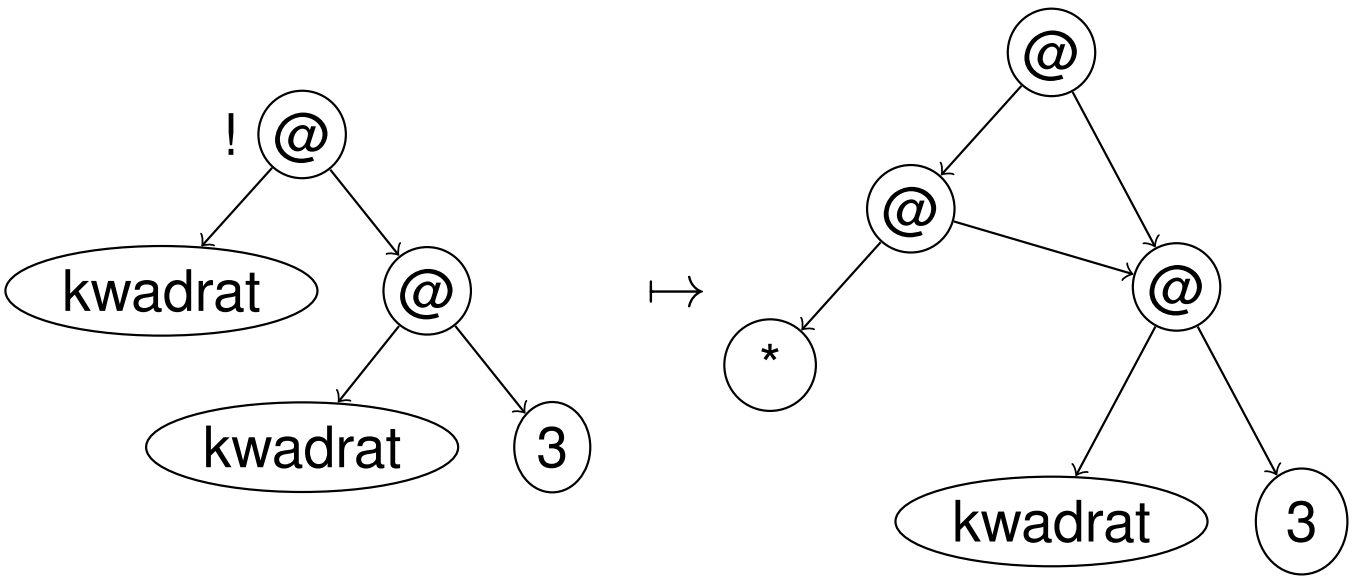
```
kwadrat x = x * x ;
main = kwadrat (kwadrat 3)
```

Redukcje będą przebiegać następująco



Redukcje grafów

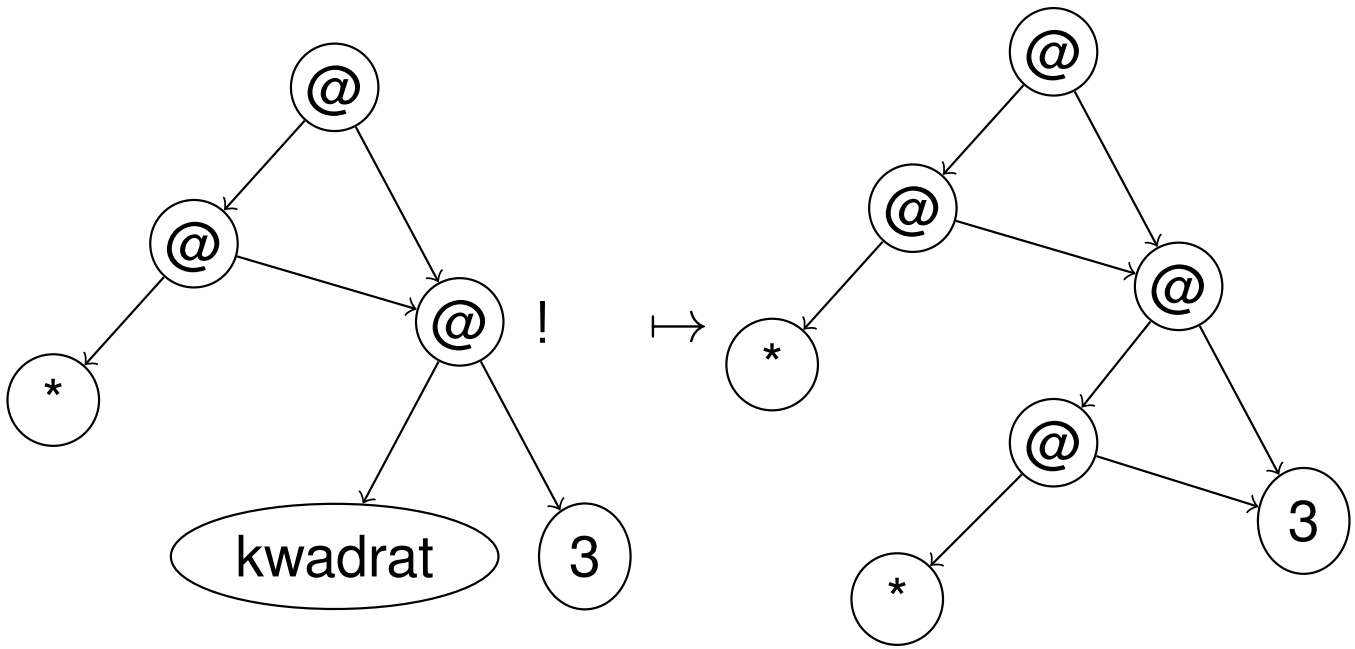
Zewnętrzną aplikację funkcji *kwadrat* zastępujemy jej grafem, zastępując w nim *x* grafem argumentu:



(! oznacza wierzchołek, w którym redukujemy, i który zostanie zastąpiony wynikiem redukcji)

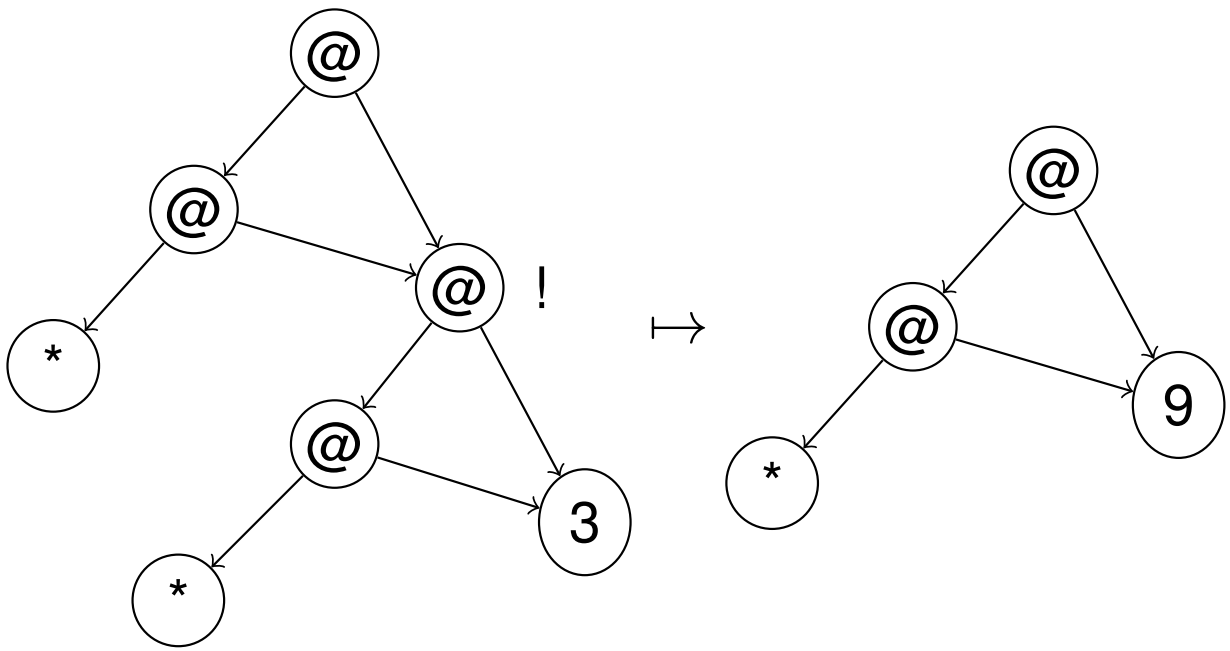
Redukcje grafów

Mnożenie można wykonać tylko na liczbach, musimy więc zredukować argumenty *:



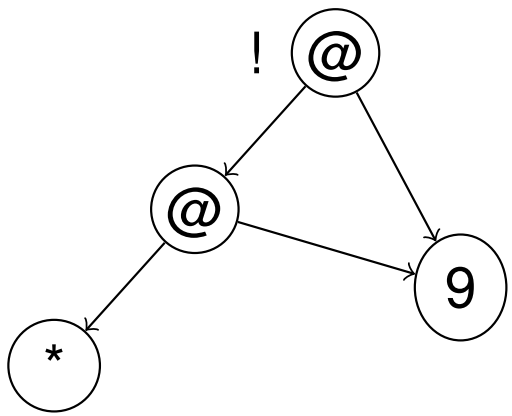
Redukcje grafów

Teraz jedyną możliwą redukcją jest wewnętrzne mnożenie:



Redukcje grafów

Ostatnia redukcja jest już bardzo prosta:



$\mapsto 81$

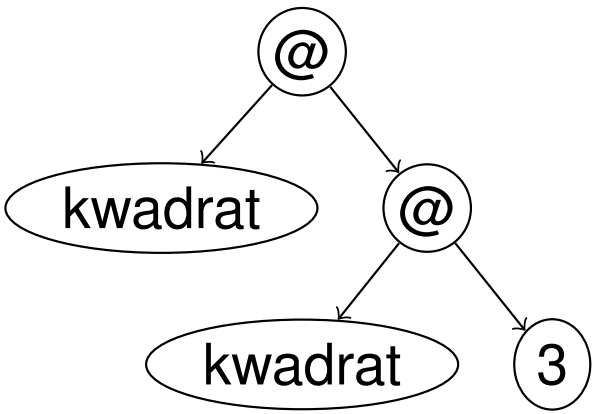
Metoda szablonów

Metoda szablonów, choć rzadko stosowana bezpośrednio (z uwagi na swą niską efektywność) leży u podstaw wielu metod implementacji języków funkcyjnych.

- dla każdej funkcji tworzymy jej szablon (graf ciała funkcji, z wolnymi "gniazdkami"dla argumentów)
- w momencie wywołania funkcji (tj. redukcji aplikacji) tworzymy wcielenie (kopię) danego szablonu z parametrami faktycznymi “włączonymi” w odpowiednie gniazdka argumentów szablonu.

Przykłady szablonów

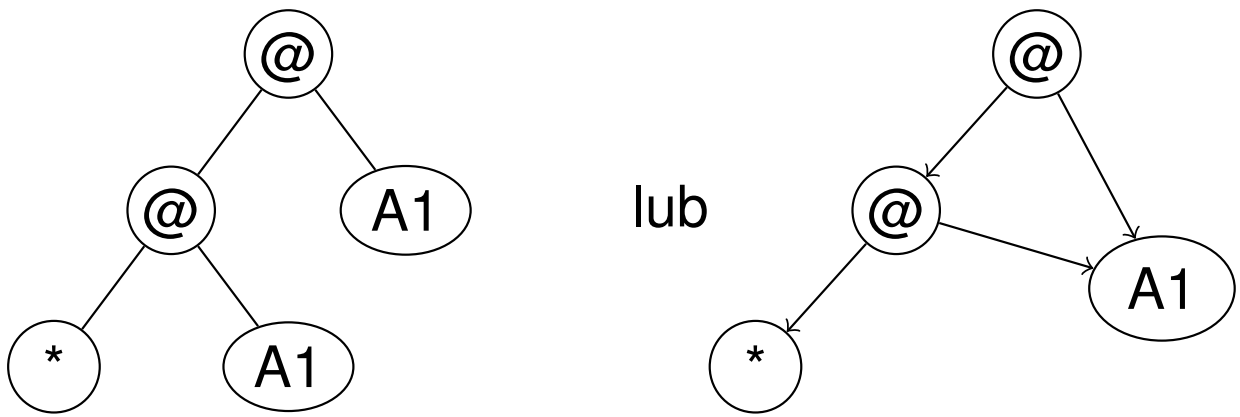
Szablon dla funkcji `main = kwadrat(kwadrat 3)`



(funkcja main nie ma argumentów)

Przykłady szablonów

Szablon dla funkcji `kwadrat x = x * x`



gdzie A1 oznacza gniazdko dla pierwszego argumentu

Nieefektywność metody szablonów

- Zastosowanie tej metody prowadzi do stworzenia raczej interpretera niż kompilatora: nie generujemy kodu w sensie ciągu instrukcji, a jedynie grafy funkcji.
- Cały proces redukcji jest realizowany przez system wykonawczy. Stąd bierze się wspomniana nieefektywność metody szablonów.
- Można usprawnić ten proces generując kod, który zbuduje graf dla funkcji. Na tym pomyśle oparta jest G-maszyna autorstwa Augustssona i Johnsona.
- Kod ten można potem przetłumaczyć na asembler (albo LLVM).

Znajdowanie następnego redeksu

Sukces/porażka obliczenia mogą zależeć od kolejności obliczeń, np. `const 10 undefined`

Strategia lewostronna: wybierz skrajny lewy redeks w drzewie wyrażenia.

Jeżeli jakaś kolejność redukcji prowadzi do sukcesu, to strategia lewostronna też. Możemy ją zrealizować następująco:

1. Zaczynając od korzenia, podążamy lewą gałęzią drzewa aż do napotkania superkombinatora. Zwykle w trakcie tego procesu zapamiętujemy odwiedzone wierzchołki na stosie. Proces ten nazywany jest *rozwijaniem grzbietu*, a jego ślad na stosie *grzbietem*.
2. Sprawdzamy ilość argumentów superkombinatora i cofamy się o taką ilość kroków w górę drzewa. Napotkany węzeł jest następnym redeksem.

G-maszyna

G-maszyna (czyli maszyna grafowa) składa się ze sterty przechowującej grafy, oraz stosu, na którym przechowywane są wskaźniki do sterty oraz stałe.

W momencie wywołania funkcji na stosie leży grzbiet (alternatywnie: wskaźniki do argumentów i redeksu).

Kod funkcji buduje graf dla wcielenia funkcji i dokonuje kolejnej redukcji.

Sercem G-maszyny są instrukcje związane z budowaniem grafów i ich redukcją. Do tej kategorii należą m.in. instrukcje:

- PUSH n — połóż na wierzchołku stosu n-ty (względem bieżącego wierzchołka stosu) argument
- PUSHGLOBAL f — połóż na stosie adres funkcji f
- PUSHINT n — połóż na stosie stałą całkowitą n
- MKAP — zbuduj węzeł aplikacji (używając dwóch pierwszych elementów stosu)

G-maszyna

- SLIDE n — usuń n elementów spod wierzchołka stosu. Jeśli stos przed tą operacją zawiera elementy

$$a_0; a1; \ldots; a_n; b; c \ldots$$

to po tej operacji będzie zawierał

$$a_0; b; c \ldots$$

- UNWIND — znajdź następny redeks (rozwijając grzbiet) i zredukuj go (skacząc do kodu odpowiedniego superkombinatora).

Przykłady

Kod dla naszej przykładowej funkcji `main` będzie zatem wyglądał następująco:

```
PUSHINT 3
PUSHGLOBAL kwadrat
MKAP
PUSHGLOBAL kwadrat
MKAP
SLIDE 1
UNWIND
```

[animacja na tablicy]

Przykłady

Zaś dla funkcji `kwadrat`

```
PUSH 0 (ARG1)
PUSH 1 (też ARG1, ale stos wzrósł o 1)
PUSHGLOBAL *
MKAP
MKAP
SLIDE 2
UNWIND
```

Zauważmy, że graf budowany przez powyższy kod nie jest drzewem - zawiera dwie krawędzie prowadzące do argumentu funkcji.

Generacja kodu

Dla funkcji n-argumentowej schemat generacji kodu wygląda następująco:

- instrukcje budujące graf (łatwo je wygenerować obchodząc szablon funkcji w porządku postfiksowym)
- SLIDE n+1 (usuwanie ze stosu starego grzbietu przy zachowaniu wskaźnika do nowozbudowanego grafu)
- UNWIND (rozwinięcie grzbietu i redukcja znalezionego redeksu).

Poza podstawowymi instrukcjami G-maszyna posiada instrukcje realizujące operacje wbudowane (np. arytmetyczne) oraz inne operacje charakterystyczne dla kompilowanego języka.

Schematy generacji kodu

Schemat $\mathcal{F}$ generuje kod dla superkombinatora

$$\mathcal{F}\,[\![\, f\, x_1 \ldots x_n = e \,]\!] = \mathcal{R}[\![e]\!]\ \rho_f\ n$$

$$\rho_f = [x_1 \mapsto 0, \ldots, x_n \mapsto n-1]$$

Schemat $\mathcal{R}$ tworzy kod, który buduje graf ciała i redukuje go:

$$\mathcal{R}[\![e]\!]\ \rho\ d = \mathcal{C}\,[\![e]\!]\ \rho +\!\!+ [\texttt{SLIDE}\ \ d+1,\ \ \texttt{UNWIND}]$$

Schemat $\mathcal{C}$ generuje kod, który buduje graf wyrażenia:

$$\begin{aligned}
\mathcal{C}\,[\![\, x \,]\!]\ \rho &= [\texttt{PUSH}\ \ \rho(x)] \\
\mathcal{C}\,[\![\, f \,]\!]\ \rho &= [\texttt{PUSHGLOBAL}\ \ \texttt{f}] \quad (f \notin \rho) \\
\mathcal{C}\,[\![\, i \,]\!]\ \rho &= [\texttt{PUSHINT}\ \ \texttt{i}] \\
\mathcal{C}\,[\![\, e_0\, e_1 \,]\!]\ \rho &= \mathcal{C}\,[\![\, e_1 \,]\!]\ \rho +\!\!+ \mathcal{C}\,[\![\, e_0 \,]\!]\ \rho^{+1} +\!\!+ [\texttt{MKAP}]
\end{aligned}$$

gdzie $\rho^{+n}(x) = \rho(x) + n$.

Leniwe obliczenia

Zauważmy, że opisana powyżej maszyna nie realizuje w pełni paradygmatu leniwych obliczeń: co prawda argumenty są obliczane dopiero kiedy potrzeba, mogą jednak być obliczne wielokrotnie. Aby temu zaradzić (i zwiększyć sprawność maszyny) musimy wprowadzić jeszcze dwie operacje (zamiast SLIDE):

- UPDATE n — zastąp wierzchołek grafu wskazywany przez $(n+1)$-szy element stosu przez wierzchołek na szczycie stosu;
  Uwaga: to zmienia graf, nie stos!
- POP n — zdejmij n elementów ze stosu.

Leniwe obliczenia

W tej wersji epilog funkcji n-argumentowej wygląda następująco:

```
UPDATE n
POP n
UNWIND
```

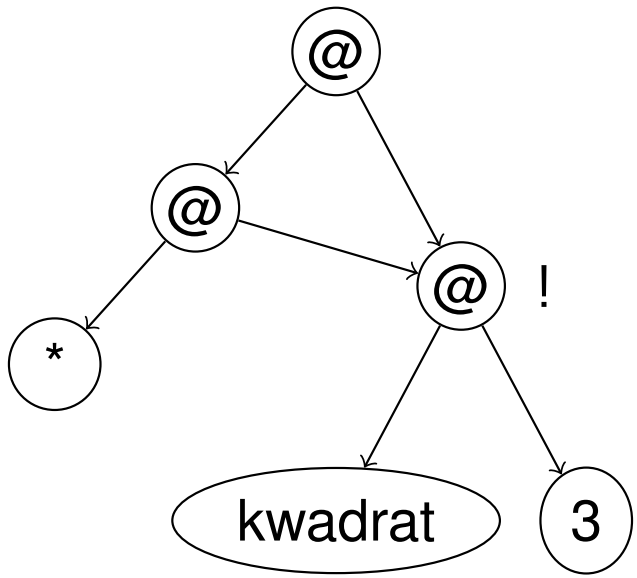
Dokładniej, musimy zmienić schemat $\mathcal{R}$:

$$\mathcal{R}[\![e]\!]\ \rho\ n = \mathcal{C}[\![e]\!]\ \rho +\!\!+ [\texttt{UPDATE n, POP n, UNWIND}]$$

[przykład na tablicy]

Operacje wbudowane

Wróćmy na chwilę do naszego przykładu
`kwadrat(kwadrat 3)`:



Jak zauważyliśmy, mnożenie możemy wykonać tylko na liczbach, więc najpierw musimy zredukować jego argumenty. Ale nasza maszyna tego nie uwzględnia. . .

Operacje wbudowane

Musimy wprowadzić nową instrukcję — EVAL, która:

- zapamieta bieżącą kontynuację (kontekst) obliczeń — wskaźnik kodu i stosu,
- obliczy wartość wyrażenia na szczycie stosu,
- powróci do zapamiętanej kontynuacji.

Kontynuacje odkładane są na dodatkowym (meta-)stosie, zwanym *składem* (dump). W praktyce można użyć stosu maszynowego.

Musimy też zmodyfikować instrukcję UNWIND, tak aby w wypadku napotkania stałej całkowitej wyjmowała zachowany kontekst ze składu, kładąc na wierzchu tę stałą — czyli zachowywała się podobnie do `ireturn` w JVM.

Operacje wbudowane — generacja kodu

Wprowadzamy nowy schemat kompilacji: $\mathcal{E}$, który generuje kod obliczający wyrażenie do postaci normalnej (WHNF):

$$\mathcal{E}\,[\![i]\!]\;\rho = [\texttt{PUSHINT i}]$$
$$\mathcal{E}\,[\![e_0 * e_1]\!]\;\rho = \mathcal{E}\,[\![e_1]\!]\;\rho +\!\!+ \mathcal{E}\,[\![e_0]\!]\;\rho^{+1} +\!\!+ [\texttt{MUL}]$$
$$\mathcal{E}\,[\![e]\!]\;\rho = \mathcal{C}\,[\![e]\!]\;\rho +\!\!+ [\texttt{EVAL}]$$

Zauważmy też, że do kodu superkombinatora wchodzimy zawsze z intencją zredukowania go, zatem schemat $\mathcal{R}$ zmodyfikujemy nastepująco:

$$\mathcal{R}[\![e]\!]\;\rho\;d = \mathcal{E}\,[\![e]\!]\;\rho +\!\!+ [\texttt{UPDATE d, POP d, UNWIND}]$$

Przykład

Kod dla funkcji `kwadrat x = x * x`

```
PUSH 0
EVAL
PUSH 1
EVAL
MUL
UPDATE 1
POP 1
UNWIND
```

(drugi EVAL jest redundantny — sprytniejszy generator kodu mógłby to wykryć)

Obliczenia leniwe i (nad)gorliwe

W strategii lewostronnej obliczenia wykonywane są dopiero wtedy gdy są potrzebne. Taki model nazywamy *leniwym* (lazy); jest on używany m.in. w Haskellu. Na przykład

```
const x y = x
main = const 0 (1/0)
```

obliczy się w tym modelu poprawnie, gdyż `1/0` nie zostanie nigdy obliczone.

Z kolei w językach rodziny ML używany jest model *gorliwy* (eager) — argumenty obliczane są przed wywołaniem funkcji.

Możliwe jest mieszanie tych dwóch modeli i obliczanie jednych wyrażeń w kontekście gorliwym (schemat $\mathcal{E}$), innych zaś w leniwym (schemat $\mathcal{C}$).

Operacje wbudowane (np. $+$) są zwykle gorliwe.

Struktury danych

Dla reprezentacji algebraicznych struktur danych (jak listy), wprowadzamy nowy rodzaj węzła —konstruktor:

$$Cons(tag, args\ldots)$$

na przykład dla list

$$[\,] = Cons(0)$$

$$(x : xs) = Cons(1, x, xs)$$

oraz instrukcje:

- `PACK t n` — zdejmuje n elementów ze stosu i opakowuje z etykietą t.
- `SPLIT` — szczyt stosu a wskazuje na $Cons(a_1, \ldots, a_n)$, zastepujemy go przez $a_1, \ldots, a_n$.
- `CASEJUMP [t1 -> kod1,...]` — skacze do kodu odpowiadającego etykiecie konstruktora wskazywanego przez szczyt stosu.

Przykład CASEJUMP

```
length xs = case xs of {
  [] -> 0; (:) y ys -> 1 + length ys }
```

```
PUSH 0
EVAL
CASEJUMP [
   0 -> [PUSHINT 0]
   1 -> [ SPLIT 2, PUSHGLOBAL length, MKAP
          EVAL, PUSHINT 1, ADD, SLIDE 2]]
UPDATE 1
POP 1
UNWIND
```

Schematy generacji kodu

Niech $C_{t,n}$ oznacza n-argumentowy konstruktor o etykiecie t.

$$\mathcal{E}\,[\![\texttt{case}\ e\ \texttt{of}\ alts]\!]\,\rho = \mathcal{E}\,[\![e]\!]\,\rho +\!\!+ \texttt{CASEJUMP}\ \mathcal{D}\,[\![alts]\!]\,\rho$$

$$\mathcal{D}\,[\![alt_1, \ldots, alt_n]\!]\,\rho = [\mathcal{A}\,[\![alt_1]\!]\,\rho, \ldots, \mathcal{A}\,[\![alt_n]\!]\,\rho]$$

$$\mathcal{A}\,[\![C_{t,n}\ x_1 \ldots x_n \to e]\!]\,\rho = t \to [SPLIT\ n] +\!\!+ \mathcal{E}\,[\![e]\!]\,\rho' +\!\!+ [SLIDE\ n]$$

$$\rho' = \rho^{+n}[x_1 \mapsto 0 \ldots x_n \mapsto n-1]$$

Kontruktory są domyślnie leniwe (a $C\ \vec{e}$ jest w WHNF):

$$\mathcal{E}\,[\![C_{t,n}\ e_1 \ldots e_n]\!] = \mathcal{C}\,[\![e_n]\!]\,\rho^{+0} +\!\!+ \ldots \mathcal{C}\,[\![e_1]\!]\,\rho^{n-1} +\!\!+ [PACK\ t\ n]$$

Trzeba tylko zapewnić, że konstruktory są zawsze nasycone, tzn nie są częściowo aplikowane — łatwe, dodajemy odpowiednie λ-abstrakcje.
Podobnie dla `case` w leniwym kontekście.

Inne maszyny

- Zauważmy, że G-maszyna poswięca dużo czasu na operację rozwijania grzbietu.
- Pojedyncza instrukcja UNWIND reprezentuje w istocie dość złożoną operację.
- Niektóre nowsze maszyny wirtualne (takie jak Three Instruction Machine (TIM) [Fairbairn,Wray] czy Spineless Tagless G-machine (STG) [Peyton Jones] zastępują operację rozwijania grzbietu przez efektywniejsze (choć bardziej skomplikowane) mechanizmy.

Lambda-lifting

Lambda-lifting jest transformacją programu do postaci superkombinatorów czyli funkcji, które nie korzystają za zmiennych nielokalnych, a jedynie ze swoich argumentów. Możemy to osiągnąć używając dwóch operacji:

- przydawanie funkcji dodatkowych argumentów przekazujących wartości zmiennych nielokalnych;
- podnoszenie (lifting) funkcji lokalnych na poziom globalny.

Na przykład funkcja

```
f xs y = map (\z -> h z y) xs
```

zostanie przetransformowana do zbioru superkombinatorów:

```
f xs y = map (g y) xs
g y z = h z y
```

Lambda-lifting

Transformacja ta realizowana jest następująco:

- dla każdej funkcji lokalnej obliczamy jej zbiór zmiennych wolnych (nielokalnych);
- tworzymy dla niej nowy superkombinator mający jako argumenty pierwotne argumenty funkcji oraz jej zmienne nielokalne;
- wystąpienie tej funkcji lokalnej zastępujemy odpowiednim zastosowaniem utworzonego superkombinatora.

Koniec

I to by było na tyle. . .