# Verslag 1: Van verrijkte $\lambda$-calculus naar $\lambda$-calculus

Koen Pauwels

16 maart 2016

## 1 Inleiding

De twee dominante theoretische modellen voor computatie zijn Turing machines en de $\lambda$-calculus. Waar Turing machines (via Von Neumann machines) de conceptuele basis vormen voor de imperatieve programmeertalen, komen de functionele talen voort uit $\lambda$-calculus.

Wanneer we spreken over compilatie van functionele programmeertalen, betekent dit voor dit project dat we een programma in een high-level functionele taal (in ons geval een subset van Haskell), voorgesteld als een reeks van definities en een expressie om te reduceren, omzetten in een $\lambda$-calculus expressie. Deze expressie zal vervolgens gereduceerd worden door een graafreductiealgoritme. Het gaat dus slechts over een gedeeltelijke compilatie aangezien we de compilatie niet verderzetten tot we machine-instructies hebben (dit is in principe mogelijk maar gaat voorbij de schaal van dit project).

Het vertalen van een programma in Haskell naar $\lambda$-calculus is een complex proces dat we liefst in kleinere stappen onderverdelen. We merken op dat de functionele talen in grotere mate dan de imperatieve talen syntactische variaties op elkaar zijn met betrekkelijk weinig semantische verschillen. We kunnen dus een taal opstellen die een semantische kern vormt voor vrijwel alle functionele talen, maar die toch een rijkere structuur heeft dan de $\lambda$-calculus. We kunnen dan Haskell door middel van betrekkelijk eenvoudige syntactische transformaties omzetten in deze *verrijkte $\lambda$-calculus*, waarna we de complexere expressies van de verrijkte vorm stap voor stap omzetten naar de gewone vorm.

## 2 Syntax en parsing

**Syntax van $\lambda$-calculus** De $\lambda$-calculus heeft een zeer eenvoudige syntax. Een eenvoudige syntax voor expressies is

1. Een constante, of
2. Een variabele, of
3. Een applicatie van de vorm "<expressie><expressie>".
4. Een abstractie van de vorm "$\lambda$ <variabele>.<expressie>".

Haakjes worden gebruikt om ambiguïteit te vermijden. We voegen nog enkele regels toe voor betere leesbaarheid en bruikbaarheid.

1. Het lambda symbool wordt vervangen door backslash.
2. Applicatie is links-associatief, dus

   ```
   (a b) c = a b c
   ```

3. Abstracties strekken zich zo ver mogelijk uit naar rechts als mogelijk.

   ```
   \x.(x y z) = \x.x y z
   ```

4. Geneste abstracties kunnen worden genoteerd als een abstractie met meerdere argumenten.

   ```
   \x.\y.E = \x y.E
   ```

Deze extra regels hebben enkel betrekking op de notatie van de expressies. Intern gebruiken we de eenvoudigst mogelijke voorstelling voor $\lambda$-calculus.

```
data Expr = Var    Symbol
          | Const  Constant
          | App    Expr       Expr
          | Abstr  Symbol     Expr
```

**Syntax van verrijkte $\lambda$-calculus** De syntax van de verrijkte $\lambda$-calculus is iets complexer.

```
<exp> ::= <constant>
      |   <variable>
      |   <exp> <exp>
      |   \<pattern>.<exp>
```

```
|   let <pattern> = <exp>,
        ...
        <pattern> = <exp>
    in <exp>
|   letrec <pattern> = <exp>,
           ...
           <pattern> = <exp>
    in <exp>
|   <exp> [] <exp>
|   case <variable> of <pattern> -> <exp>;
                        ...
                        <pattern> -> <exp>;
```

De betekenis van de meeste nieuwe elementen is vanzelfsprekend: let en letrec introduceren lokale scope (de laatste staat recursieve definities toe), case expressies selecteren een expressie op basis van het patroon waarmee de variabele onder analyse overeenkomt. Enkel de [] operator ("*fatbar*") is niet onmiddellijk duidelijk. De operator is een builtin die wordt gebruikt bij pattern matching. Stel dat we een Haskell functie $f$ hebben met n patronen:

f p1 = E1
...
f pn = En

We transformeren dit naar de volgende verrijkte *lambda*-calculus expressie (waar x een nieuwe variabele is die in geen enkele E-expressie vrij voorkomt):

```
f = \x.(  ((\p1.E1) x)
       [] ((\p2.E2) x)
       ...
       [] ((\pn.En) x)
       [] ERROR)
```

We kunnen dit lezen als "probeer eerst x te matchen met p1, als dat lukt, evalueer E1 (met de correcte binding), anders, probeer x te matchen met p2, ...; als alle patterns falen, return de builtin waarde ERROR".

De extra regels van de gewone $\lambda$-calculus (links-associativiteit e.d.) hebben ook op deze grammatica betrekking.

**Parsing van $\lambda$-calculus** Voor het parsen heb ik de monadische parser combinator library *Parsec* gebruikt. Dit is een populaire aanpak binnen functioneel programmeren om recursive descent parsers te schrijven. Parsers worden voorgesteld als functies, en er worden functies van hogere orde gedefinieerd

(combinators) om grammaticale constructies zoals sequenties, keuzes en repetities te implementeren. Het feit dat deze parsers monadisch zijn, maakt het makkelijk om complexe parsers op te bouwen door compositie van eenvoudigere parsers. Om een idee te geven van hoe zo'n parser werkt, doorlopen we een kort voorbeeldje.

```
let n = 5, m = 10 in E
```

De meest algemene parser, de *expr* (expression) parser gaat een aantal mogelijkheden proberen:

```
expr =      (abstr       <?> "abstraction"       )
       <|> (letExpr      <?> "let expression"    )
       <|> (letrecExpr   <?> "letrec expression" )
       <|> (caseExpr     <?> "case expression"   )
       <|> (try subexpr  <?> "subexpression"     )
       <|> (constExpr    <?> "constant"          )
       <|> (varExpr      <?> "variable"          )
```

De $< | >$ operator geeft aan dat hier een keuze tussen beperktere parsers mogelijk is. De expr parser stopt bij de eerste parser die erin slaagt om de expressie te parsen. Eerst wordt geprobeerd de expressie als een abstractie te parsen, maar dat gaat onmiddellijk mislukken omdat er geen \ staat om de abstractie aan te kondigen. Vervolgens wordt de *letExpr* parser aangeroepen. Die ziet er zo uit:

```
-- Parses a simple let expression.
-- A let expression with multiple comma-separated definitions is
-- translated to a nesting of single-definition let expressions.
letExpr = do m_reserved "let"
             ds <- sepBy1 definition (m_reservedOp ",")
             m_reserved "in"
             e <- expr
             return $ foldr ((.) . LetExpr) id ds e
```

Hier wordt gebruikt gemaakt van monadische *do* notatie, die makkelijk sequentieel gelezen kan worden: lees eerst het gereserveerde woord "let" in, daarna een lijst van definities, minstens 1 lang, gescheiden door komma's, daarna het gereserveerde woord "in", gevolgd door terug een expressie. De lijst van geparsete definities en de geparsete expressie worden onthouden en omgezet in een set geneste let-expressies. De definition parser is hier nog niet gedefinieerd maar dat zou ons te ver leiden. Het volledige algoritme is te vinden in *EnrichedLambdaParse.hs*.

# 3 Verarmingsalgoritme

De omzetting van verrijkte naar gewone $\lambda$-calculus, het verarmingsalgoritme, vormde het grootste deel van het werk totnogtoe. In essentie is het een case analyse waarbij we steeds kijken naar de structuur van de toplevel expressie, waarna we een transformatie toepassen zodat we ofwel een gewone $\lambda$-expressie, ofwel een eenvoudigere verrijkte expressie krijgen. In de meeste gevallen moeten we recursief in de structuur afdalen om de deelexpressies eerst om te zetten voordat we de toplevel expressie vereenvoudigen.

```
-- Transform the enriched lambda calculus to ordinary lambda calculus.
impoverish :: EL.Expr TypeDef -> L.Expr
```

Het type *EL.Expr TypeDef* stelt een verrijkte expressie voor waar elke type constructor een referentie bijhoudt naar de typedefinitie waarin hij gedefinieerd werd. Verder zal duidelijk worden waarom dit nodig is.

## 3.1 Constanten, variabelen, applicaties

Dit zijn de eenvoudigste gevallen. Constanten en variabelen worden niet aangepast, en voor applicaties moeten we enkel de verarming toepassen op de deelexpressies.

```
impoverish (EL.VarExpr v)     = L.Var v
impoverish (EL.ConstExpr k)   = L.Const k
impoverish (EL.AppExpr e f)   = L.App (impoverish e) (impoverish f)
```

## 3.2 Abstracties

Bij abstracties van de vorm

```
\p.E
```

onderscheiden we 4 gevallen, afhankelijk van de structuur van $p$.

1. Als $p$ een eenvoudige variabele is, is de vertaling gewoon een kwestie van E te verarmen.

2. Als $p$ een constant patroon $k$ is (bijvoorbeeld “5”), moeten we nagaan dat het argument gelijk is aan de constante. Als dit het geval is voeren we het functielichaam uit, en anders aangeven dat het pattern matchen faalde. We vertalen de abstractie dus naar (waarbij v een nieuwe variabele is die niet vrij voorkomt in E, en E de verarmde versie van de originele E).

   ```
   \v.IF (= k v) E FAIL
   ```

3. Wanneer $p$ een product-constructor patroon met ariteit $r$ is, dan hebben we een expressie van de vorm:

   ```
   \(t p1 ... pr).E)
   ```

   Product-constructors zijn constructors voor een type met slechts 1 constructor. We hoeven dus niet na te gaan of de constructor van het patroon en die van het argument overeenkomen, aangezien we ervan uit mogen gaan dat ons programma al in een eerder fase getypecheckt is. Voor dit geval vinden we eenvoudigweg een nieuwe functie voor elke productconstructor $t$ uit: UNPACK-PRODUCT-$t$. De expressie wordt dan vertaald naar

   ```
   UNPACK-PRODUCT-t (\p1...\pr.E)
   ```

   UNPACK-PRODUCT-$t$ is een functie met twee parameters, een functie en een gestructureerd object. De functie wordt toegepast op de velden van de datastructuur op een lazy manier: wanneer de functie een argument niet gebruikt, wordt het ook niet geëvalueerd.

   ```
   UNPACK-PRODUCT-t f a = f (SEL-t-1 a) ... (SEL-t-r a)
   ```

4. Het laatste geval is wanneer $p$ een som-constructorpatroon is, dwz een constructorpatroon waarvan er meerdere constructors bestaan binnen het type. We moeten dus wel nagaan of de constructor van de functieparameter overeenkomt met de constructor van het meegegeven argument. Ook hier gaan we een nieuwe functie invoeren, UNPACK-SUM-$s$ (met $s$ de som-constructor). De semantiek hiervan is gelijkaardig aan die van UNPACK-PRODUCT, maar de returnwaarde is FAIL als de constructors niet overeenkomen.

Het feit dat we onderscheid moeten kunnen maken tussen som en product constructor patterns verklaart onmiddellijk waarom we typeinformatie nog steeds moeten bijhouden terwijl we in de verrijkte $\lambda$-calculus werken.

## 3.3 let en letrec

Eenvoudige let-expressies die geen pattern matching doen kunnen worden vertaald met de volgende eenvoudige transformatie:

```
let v = B in E  <=>  (\v.E) B
```

We staan echter toe dat de definities pattern-matchen. We onderscheiden hier twee gevallen: *weerlegbare* (of *refutable*) en *onweerlegbare* (of *irrefutable*) patronen.

De laatste categorie is de makkelijkste: hier is het niet mogelijk dat het pattern matchen faalt. We voeren dus gewoon een transformatie uit die ervoor zorgt dat de juiste waarden aan de juiste delen van het patroon gebonden worden, waarna we het verarmingsalgoritme weer uitvoeren op het resultaat. Bijvoorbeeld

```
let (PAIR x y) = b in E  <=> let v = B in (let x = SEL-PAIR-1 v,
                                                y = SEL-PAIR-2 v
                                            in E)
```

Onweerlegbare letrecs zetten we eerst om in onweerlegbare lets met behulp van de ingebouwde Y-combinator.

```
letrec v = B in E  <=>  (let v = Y(\v.B) in E)
```

Weerlegbare let(rec)s zijn wat complexer. We gaan deze eerst omzetten in onweerlegbare let(rec)s, en dan het resultaat verarmen zoals hierboven staat beschreven. Omdat een weerlegbaar patroon kan falen, moeten we een *conformaliteitstest* uitvoeren. We gebruiken hiervoor de eerder gedefinieerde machinerie van pattern-matching abstracties (die geven FAIL terug wanneer de pattern match faalt). Een voorbeeld, waarbij CONS een constructor is voor een lijst (de andere lijst constructor is die van de lege lijst, NIL):

```
let (CONS y ys) = B in E
```

wordt

```
let (PAIR y ys) = ((((\(CONS y ys).PAIR y ys) B) [] ERROR)
in E
```

### 3.4 case-expressies

Voor het transformeren van case-expressies introduceren we weer een functie per somtype, de CASE-T functie. Voor producttypes is dit niet nodig, de transformatie daar is gelijkaardig aan die van pattern-matching abstracties. Als LEAF en BRANCH constructors zijn van een "tree" type, dan kunnen we de volgende functie

```
reflect = \t.case t of
                (LEAF n)       -> LEAF n;
                (BRANCH t1 t2) -> BRANCH (reflect t2) (reflect t1);
```

vertalen als

```
reflect = \t.case-tree
                t
                (UNPACK-SUM-LEAF (\n.LEAF n) t)
                (UNPACK-SUM-BRANCH
                        (\t1.\t2.BRANCH (reflect t2) (reflect t1)))
```

De case-tree functie is zo gedefinieerd dat er precies 1 expressie moet kunnen worden geëvalueerd voor elke constructor van het somtype. De functie selecteert de correcte expressie aan de hand van de *structure tag* die de constructor identificeert.

## 3.5 De [] operator en FAIL

De [] operator is niet meer dan syntactische suiker.

```
E1 [] E2  <=>  FATBAR E1 E2
```

FATBAR is een builtin functie die haar eerste argument teruggeeft wanneer dat niet tot FAIL (of bottom) evalueert, en anders haar tweede argument. We gaan zoveel mogelijk proberen om FAIL en [] te elimineren in de stap van de vertaling van Haskell naar verrijkte *lambda*-calculus.