Individueel project

Compiler voor een zuiver functionele taal

Koen Pauwels

23 juni 2016

Introductie

Persoonlijke doelstellingen

- Niet-triviaal project in Haskell
- Beter begrip van werking compiler, ihb lazy evaluation

Overzicht compiler/interpreter

1. Gebruiker geeft programma in in verrijkte LC
2. Programma wordt vertaald naar gewone LC (boomstructuur)
3. Boomstructuur wordt geherinterpreteerd als graaf
4. Uitvoering programma door middel van graafreductie

Lambdacalculus

1. Een constante, of
2. Een variabele, of
3. Een applicatie van de vorm “<expressie><expressie>”.
4. Een abstractie van de vorm “λ <variabele>.<expressie>”.

Verrijkte lambdacalculus

```
<exp> ::= <constant>                                          Constants
        |   <variable>                                        Variables
        |   <exp> <exp>                                       Applications
        |   λ <pattern> . <exp>                               Lambda abstractions
        |   let    <pattern> = <exp> in <exp>                 Let-expressions
        |   letrec <pattern> = <exp>                          Letrec-expressions
                   . . .
                   <pattern> = <exp>
            in <exp>
        |   <exp> [] <exp>                                    Fat bar
        |   case <variable> of                                Case-expressions
            <pattern>  ⇒  <exp>
            . . .
            <pattern>  ⇒  <exp>

<pattern> ::= <constant>                                      Constant patterns
            |   <variable>                                    Variable patterns
            |   <constructor> <pattern>                       Constrnctor patterns
                              . . .
                              <pattern>
```

Verrijkte lambdacalculus naar lambdacalculus

Voorbeeld onweerlegbaar patroon

```
data Pair a b = Pair a b

let second = \(PAIR x y).y
in second (PAIR 1 2)
```

wordt

```
UNPACK–PRODUCT–PAIR (\x.\y.y)
                    (CONSTR–PAIR 1 2)
```

wat verder wordt uitgewerkt tot

```
(\x.\y.y) (SEL–PAIR–1 (CONSTR–PAIR 1 2))
          (SEL–PAIR–2 (CONSTR–PAIR 1 2))
```

Voorbeeld weerlegbaar patroon

```
data List a = CONS a (List a) | NIL

case x of (CONS x y) -> x; (NIL) -> 0;
```

wordt

```
(\a. (UNPACK-SUM-CONS (\x.\y.x) a)
     []
     (UNPACK-SUM-NIL a))
x
```

Voorbeeld recursie

```
letrec fac = \n . IF (= n 0)
                     1
                     (* n (fac (- n 1)))
in fac 2
```

wordt

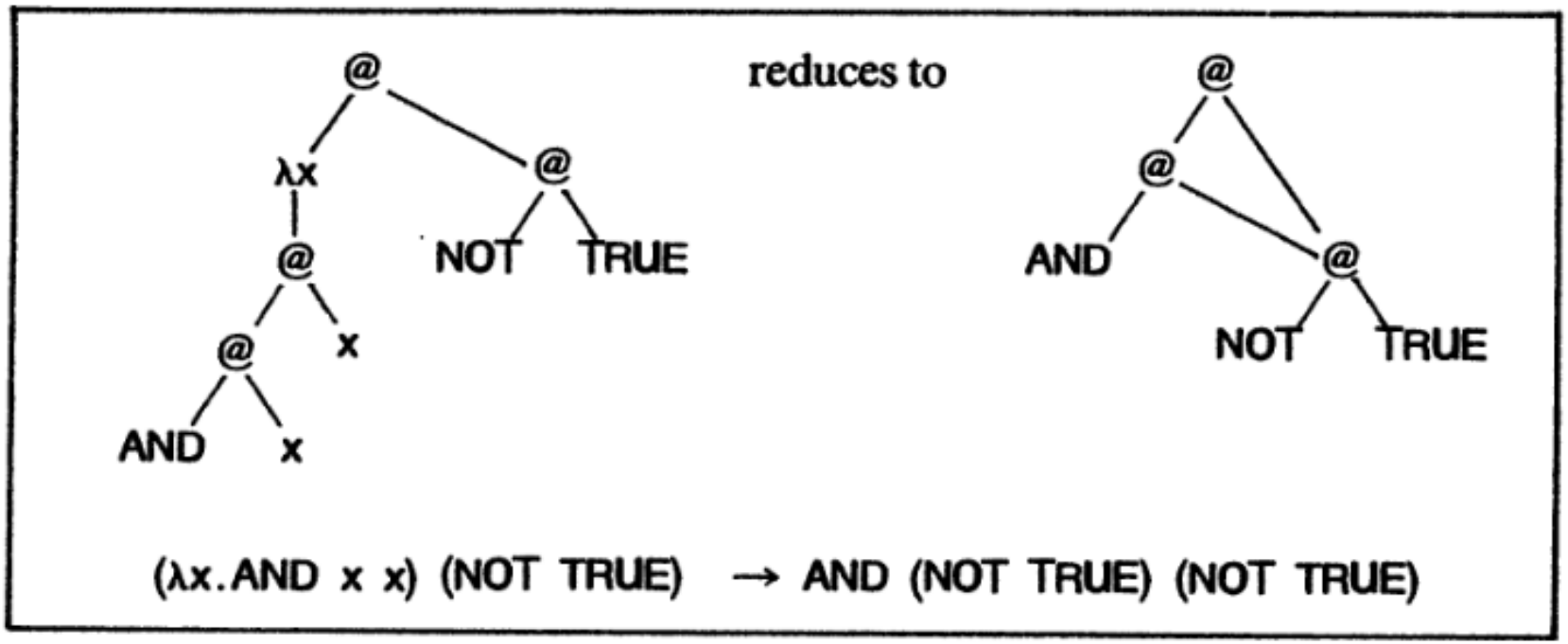
```
(\fac.fac 2) (Y (\n.IF (= n 0)
                       1
                       (* n (- n 1))))
```

Reductie

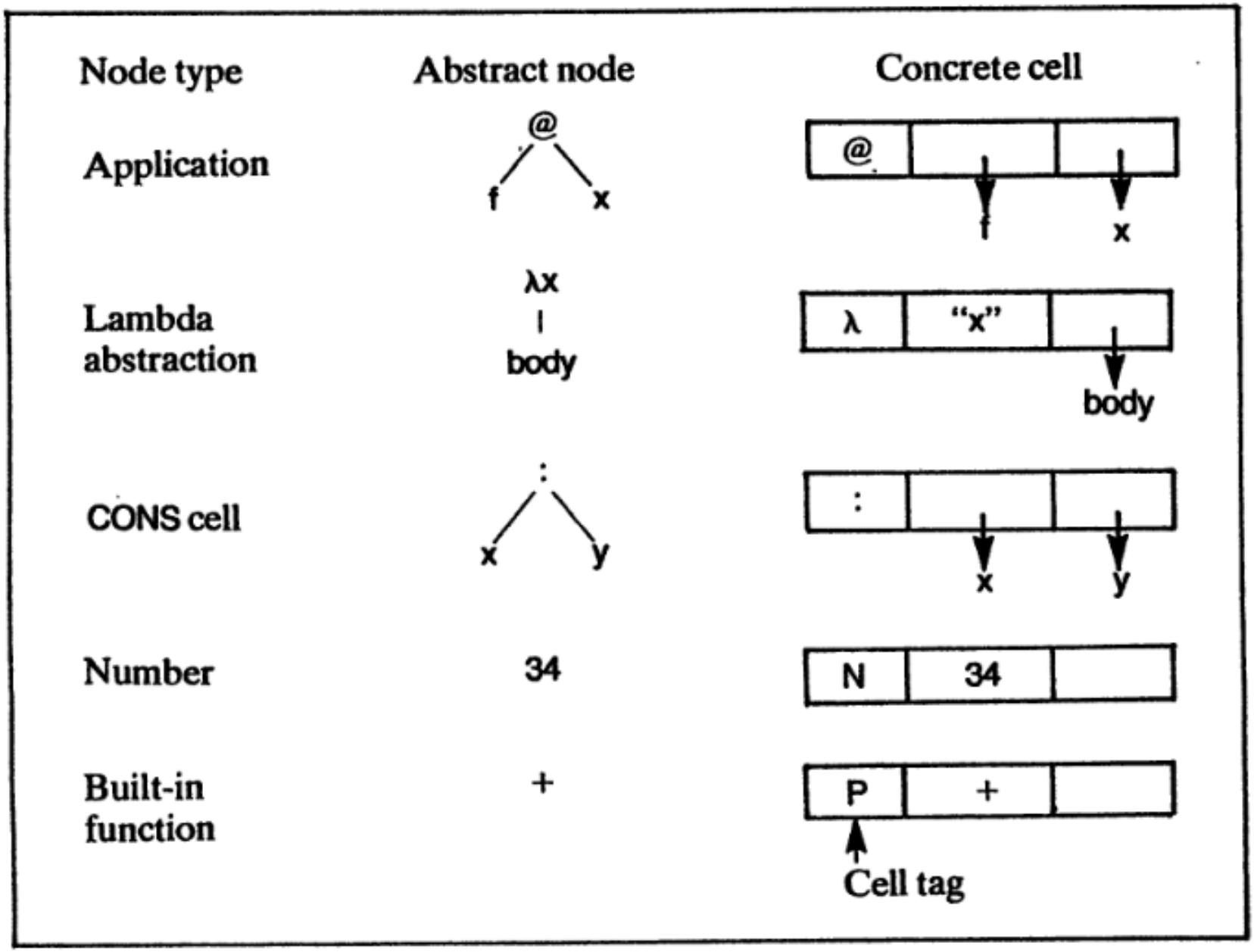
Reductie: waarom een graaf?

```
(\x.AND x x) (NOT TRUE)
-> AND (NOT TRUE) (NOT TRUE)
-> AND FALSE (NOT TRUE)
-> AND FALSE FALSE
-> FALSE
```

Reductie: waarom een graaf?



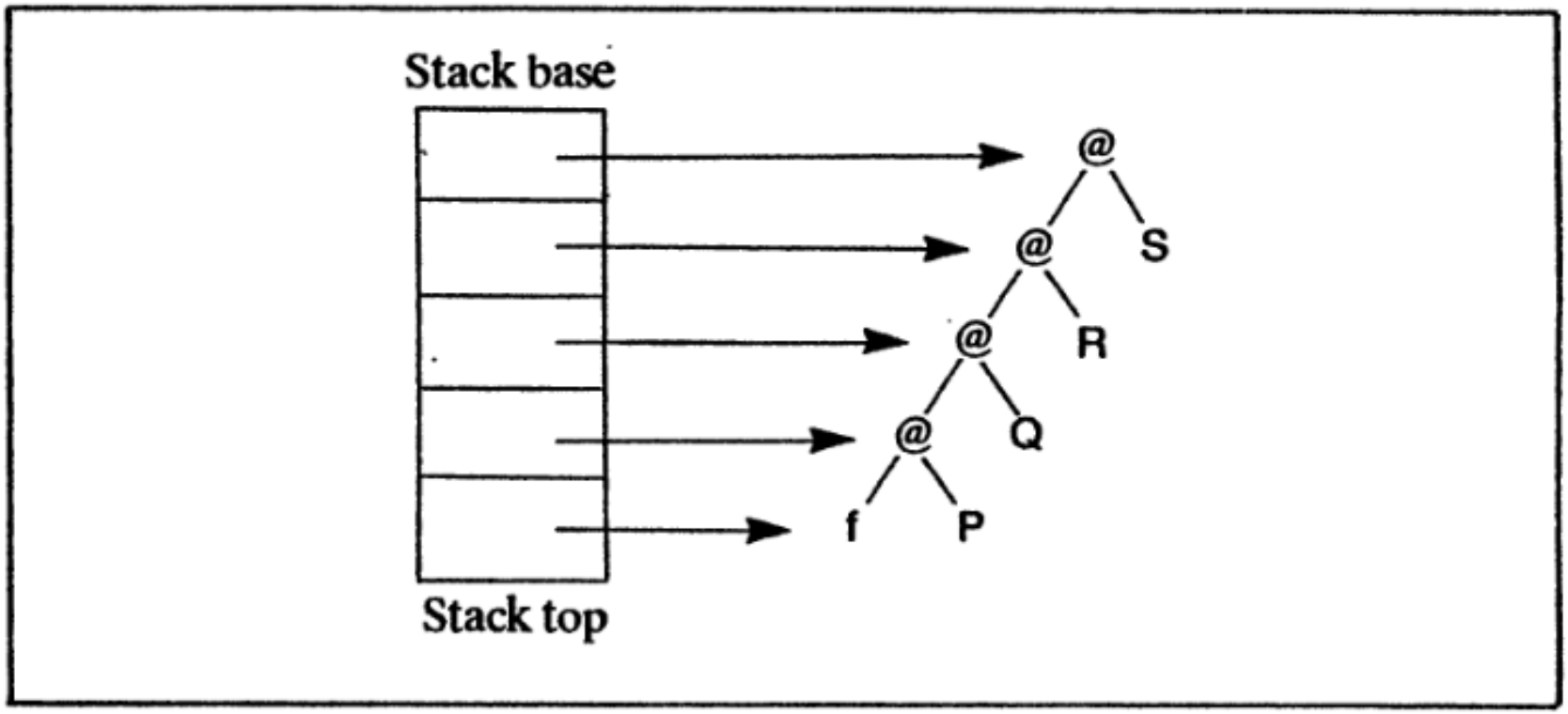
Implementatie van de graaf



Reductie gedreven door nood om output te produceren

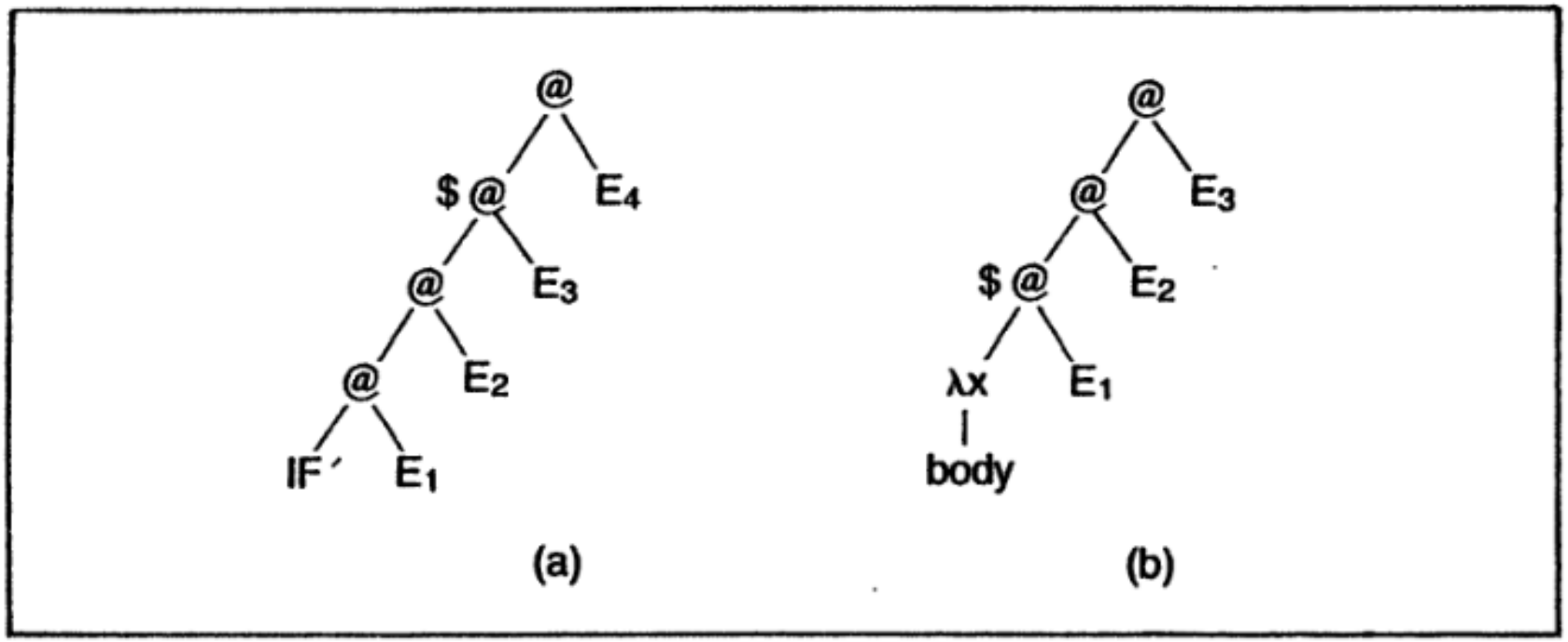
1. Printing mechanisme roept reduce functie aanknopingspunt op.
2. Die reduceert de graaf tot die zich in *weak head normal form (WHNF)* bevindt.
3. Als het resultaat een lijst is, gaat het printing mechanisme zichzelf recursief oproepen eerst op het eerste element van de lijst, en dan op de rest van de lijst. Anders wordt het resultaat onmiddellijk uitgeprint en is de huidige call van de print procedure klaar.

Hoe de volgende redex te vinden



Figuur: Illustratie van de spine stack (f P Q R S)

Hoe de volgende redex te vinden



Figuur: Voorbeelden van redex selectie (redex gemarkeerd met '$')