Verslag 1: Van verrijkte λ-calculus naar λ-calculus

Koen Pauwels

16 maart 2016

1 Inleiding

De twee dominante theoretische modellen voor computatie zijn Turing machines en de λ-calculus. Waar Turing machines (via Von Neumann machines) de conceptuele basis vormen voor de imperatieve programmeertalen, komen de functionele talen voort uit λ-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 λ-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 λ-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 λ-calculus. We kunnen dan Haskell door middel van betrekkelijk eenvoudige syntactische transformaties omzetten in deze *verrijkte λ-calculus*, waarna we de complexere expressies van de verrijkte vorm stap voor stap omzetten naar de gewone vorm.

2 Syntax en parsing

Syntax van λ-calculus De λ-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 "λ <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 λ-calculus.

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

Syntax van verrijkte λ-calculus De syntax van de verrijkte λ-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 λ-calculus (links-associativiteit e.d.) hebben ook op deze grammatica betrekking.

Parsing van λ-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 λ-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 λ-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 λ-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.

Verslag 2: Lazy evaluation door middel van graafreductie

Koen Pauwels

16 mei 2016

1 Inleiding

1.1 Waarom lazy evaluation?

In de meeste programmeertalen worden argumenten aan een functie geëvalueerd voordat de functie wordt opgeroepen. Dit noemen we een *call by value* strategie. Het is echter mogelijk dat een argument nooit wordt gebruikt in de functie. In dat geval wordt er overbodig werk geleverd.

We kunnen proberen dit overbodig werk te vermijden door de evaluatie van een argument uit te stellen totdat de waarde ervan nodig is. Dit noemen we een *call by need* of *lazy evaluation* strategie. Deze strategie is zo goed als uniek voor zuiver functionele talen. In imperatieve talen berust de betekenis van een programma op het in een correcte, voorspelbare volgorde plaatsvinden van bepaalde *side-effects*. Met *call by need* kan het erg moeilijk worden om uit te werken in welke volgorde de expressies zullen worden geëvalueerd, en dus in welke volgorde de side-effects zullen plaatsvinden.

1.2 Waarom een graafvoorstelling?

Wanneer een λ-expressie wordt ingeladen in de compiler, zal deze een boomvoorstelling van de expressie genereren. Zoals eerder vermeld worden argumenten aan functies enkel geëvalueerd wanneer de waarde nodig is. Naïeve normal-order reductie op de boomvoorstelling van de λ-expressie voldoet wel aan deze voorwaarde, maar zal vaak tot dubbel werk (en dus tot grote inefficiënties) leiden. Wanneer een parameter namelijk meerdere keren voorkomt in het lichaam van een expressie, zal elk voorval van die parameter vervangen moeten worden door een kopie van het argument. Als we het programma voorstellen als een graaf in plaats van een boom, dan kunnen we elk voorval van de parameter in

het lichaam van een λ-expressie vervangen door een *referentie* naar het argument, in plaats van een kopie ervan. Bij het evalueren van een argument zullen we ervoor zorgen dat de wortelnode van het argument wordt overschreven met het resultaat van de evaluatie. Elke referentie naar het argument verwijst dan automatisch en zonder extra kost naar het resultaat van de evaluatie.

1.3 Oriëntatie in het project

Het gehele project betreft een compiler die een zuiver functionele taal via verrijkte λ-calculus vertaalt naar gewone λ-calculus. De bekomen λ-calculus expressie wordt dan gereduceerd in een *runtime omgeving* die een graafreductie-algoritme toepast en de resultaten uitprint. Dit verslag betreft deze runtime omgeving.

2 Programmarepresentatie

De graafvoorstelling moet alle constructies van de λ-calculus ondersteunen, maar verder verwachten we ook ondersteuning voor een aantal bijzondere builtin operaties en datastructuren. In theorie is de λ-calculus voldoende om alle programma's die we wensen uit te drukken, maar in praktijk willen we onder meer de machinevoorstelling gebruiken voor getallen, machine-operaties kunnen uitvoeren op die getallen, gestructureerde datatypes kunnen gebruiken, etc.

De basis van de graafvoorstelling is de *cel*, een eenvoudige datastructuur die uit drie elementen bestaat: een *tag* die aangeeft over welk type cel het gaat, en twee *fields*. Een field kan een pointer naar een andere cel bevatten (op deze manier worden de edges van de graaf geïmplementeerd), of een waarde waarvan de betekenis afhangt van het type cel en de context.

De eerste drie celtypes implementeren de structuren van de λ-calculus, zoals beschreven in *The Implementation of Functional Programming Languages (IFPL)*: variabelen, applicaties en abstracties.

- Een `VAR` cel gebruikt slechts 1 van zijn fields. Dat veld bevat een pointer naar een string die het symbool opslaat. Er is een efficiëntere implementatie mogelijk: we zouden bij het compileren elk symbool kunnen associëren met een uniek getal, en dit getal rechtstreeks in het veld opslaan. Ik heb besloten om de originele namen te behouden om debugging niet te bemoeilijken.

- De velden van een `APP` cel zijn eenvoudigweg celpointers naar de operator en de operand van een applicatie.

Listing 1: Definitie van de Cell datastructuur

```
typedef enum {VAR, APP, ABSTR, DATA, BUILTIN, CONSTR} Tag;
union Field {
        struct Cell* ptr;
        char* sym;
        int num;
        Builtin op;
        StructuredDataTag data_tag;
        StructuredDataPtr data_ptr;
};
struct Cell {
        Tag tag;
        union Field f1;
        union Field f2;
};
```

- Een `ABSTR` cel bevat een pointer naar het parameter symbool, en een celpointer naar het lichaam van de functie.

Het boek is minder helder over welke extra tags er nodig zijn om de implementatie te vervolledigen. Mijn implementatie maakt verder nog onderscheid tussen datacellen, builtins (operatoren en constanten) en constructors.

- `DATA` cellen worden gebruikt om gegevens in op te slaan. *IFPL* lijkt onderscheid te maken tussen cellen die getallen opslaan en cellen die lijsten (`CONS` cellen) opslaan, en eventueel nog aparte structured data cellen. Ik zag geen reden om de data cellen bewust te maken van de betekenis van de gegevens die ze dragen, aangezien dit steeds af te leiden valt uit de context. Een functie die iets met de data doet, bevat genoeg informatie om de data te interpreteren. `DATA` cellen worden dus zowel gebruikt om primitieve data in op te slaan, zoals integers, of om naar gestructureerde datastructuren te verwijzen. In het eerste geval bevat de datacel enkel de waarde van de primitieve data. In het tweede geval bevat de cel een *structured data tag* en een pointer naar de datastructuur. Wanneer de datastructuur werd aangemaakt door een constructor van een somtype, is het de structured data tag die onthoudt welke constructor dit was.

- Sommige operaties kunnen efficiënter worden uitgevoerd door een `BUILTIN` operatie te voorzien in de runtime omgeving. Arithmetische en logische operatoren kunnen geïmplementeerd worden door middel van snelle machine-instructies. Conditionals en recursie kunnen ook worden uitgedrukt in λ-calculus, maar met ingebouwde `IF` en fixed point operatoren vermijden we heel wat overhead. Verder kan enkel de `SELECT` operator

velden uit een gestructureerd datatype opvragen.

Behalve operatoren zijn er ook enkele constanten die als builtins worden geïmplementeerd. Deze kunnen gezien worden als operatoren met 0 argumenten. Een voorbeeld hiervan is de `FAIL` constante, die gebruikt wordt om een gefaalde poging tot pattern matching te indiceren.

- Hoewel constructors in principe voorgesteld zouden kunnen worden door builtins, heb ik een apart `CONSTR` celtype gedefinieerd. Alle andere ingebouwde procedures hebben een vast aantal parameters per procedure: `+` heeft er steeds 2, `IF` heeft er 3, etc. Aangezien de gebruiker zijn eigen constructors kan definiëren met een arbitrair aantal velden, moet de `CONSTR` builtin omkunnen met een variabel aantal argumenten. Het leek eenvoudiger om hiervoor een apart celtype te introduceren.

3 Implementatie van het reductiealgoritme

We hebben al vastgesteld dat we een expressie pas wensen te evalueren wanneer haar waarde nodig is, maar we hebben nog geen procedure beschreven hoe we bepalen welke waarde we nodig hebben. We weten dat ons programma in elk geval uiteindelijk een zekere output moet genereren. Tot dit eind kunnen we een printing mechanisme voorzien dat het resultaat van ons programma gaat weergeven op de command line.

Bij een strikte zuivere taal zouden we het programma eerst volledig reduceren, en daarna het resultaat aan het printing mechanisme geven om zo de output te genereren.

Bij onze luie taal loopt dit anders: de print functie gaat de graafreductie aandrijven door de graaf steeds net voldoende te reduceren om het volgende element te kunnen uitprinten. Indien het resultaat van ons programma een eenvoudig geheel getal is, zien we op dit hoogste niveau niet zo veel verschil: de graaf moet volledig worden gereduceerd tot een enkele `DATA` cel die een geheel getal bevat.

Het wordt interessanter wanneer het resultaat van het programma een lijst is. Als we in een klassieke imperatieve taal een proces willen beschrijven dat een zeer lange reeks resultaten produceert, dan doen we dat gewoonlijk door middel van een lus die een resultaat produceert en onmiddellijk wegschrijft, waarna het resultaat zelf uit het geheugen verwijderd of overschreven kan worden.

In deze zuiver functionele taal hebben we deze optie niet, omdat we berekeningen en effectien niet kunnen vermengen. Het programma moet dus een lijst met alle resultaten produceren. Als het printing mechanisme echter vereist dat de volledige lijst wordt geëvalueerd voordat het begint met resultaten wegschrijven, dan betekent dit dat de volledige lijst eerst in het werkgeheugen van de computer moet worden geplaatst. Dit is duidelijk een inferieure oplossing in vergelijking met de aanpak van de imperatieve taal: het verbruik van werkgeheugen schaalt lineair met de lengte van de output in het functionele programma!

Om dit te vermijden zal de outputprocedure steeds net voldoende reduceren om 1 resultaat te kunnen wegschrijven. Wanneer een resultaat is weggeschreven, zal de node die het resultaat beschrijft worden losgekoppeld van het deel van de graaf waar we in aan het werken zijn. Als er geen andere nodes nog verwijzen naar het resultaat, zal in een runtime omgeving met geheugenmanagement dit resultaat kunnen worden verwijderd uit het geheugen.

3.1 Selecteren van de volgende te reduceren expressie

Een expressie is reduceerbaar wanneer er zich ergens in de graaf een applicatie bevindt met in het linkerlid een λ-uitdrukking, of een applicatie van een builtin op het correcte aantal argumenten voor die builtin.

Zo'n reduceerbare expressie noemen we een *redex*.

Een strikt evaluatiealgoritme zal steeds alle redexes reduceren. Met andere woorden, het brengt de graaf in *normal form*.

Zoals we al hebben vastgesteld, wensen we niet noodzakelijk alle redexes te reduceren alvorens controle terug te geven aan het printing mechanisme. Kortom, we willen normal-order reductie volgen, maar we willen dat de reductie ophoudt wanneer er geen *top-level redex* meer is. Een graaf die geen top-level redex meer heeft is in *weak head normal form*. Een expressie is in weak head normal form wanneer ze van de vorm `F E1 E2 ... En` is met $n \geq 0$, en

1. `F` is een variabele of dataobject, OF
2. `F` is een λ-abstractie of builtin operator en `(F E1 E2 ... Em)` is geen redex voor eender welke $m \leq n$.

Om even samen te vatten hoe het reductiealgoritme er momenteel uitziet:

1. Het printing mechanisme roept de reduce functie op op de knoop die het aanknopingspunt vormt van het programma.

2. De reduce functie reduceert de graaf tot die zich in *weak head normal form (WHNF)* bevindt.

3. Indien het resultaat van de reductie 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.

   Indien het resultaat van de reductie een geheel getal (of een ander primitief datatype) is, wordt dit onmiddellijk uitgeprint en is de huidige call van de print procedure klaar.

Omdat de type-informatie van de data uitgewist is tijdens runtime, moet het printing mechanisme in mijn implementatie bij de compilatie op de hoogte worden gebracht wat het type van de output zal zijn (bv "een lijst van lijsten van integers"). Wanneer de typechecker geïmplementeerd is zal dit automatisch gebeuren, nu moet de informatie nog handmatig worden meegegeven.

3.2 Hoe de volgende top-level redex te vinden

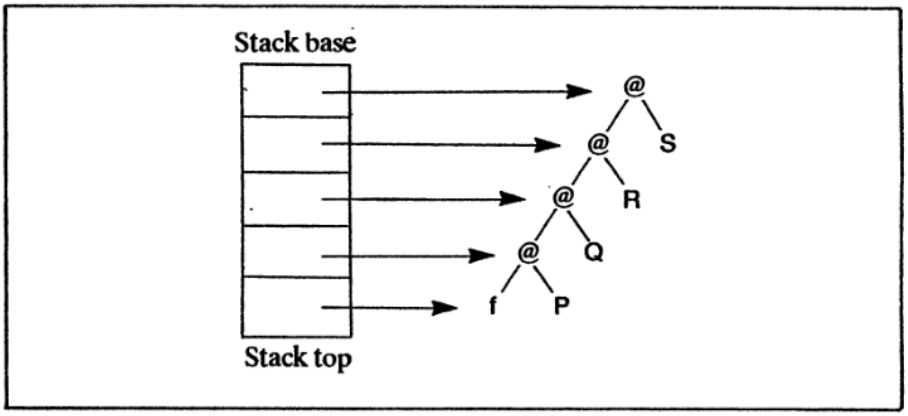
De expressie die we wensen te reduceren kan enkel van de vorm `f E1 E2 ... En` zijn. Applicatie is links-associatief: `f E1 E2 ... En` wordt gelezen als `(...((f E1) E2) ... En)`. Bij het begin van de reductie ziet het reductiealgoritme enkel de hoogste applicatie (`((...) En)`). Eerst moet het `f` zoeken, de operator van de reductie, door de n applicatienodes te doorlopen (steeds het linkerlid kiezend). Deze aaneenschakeling van applicatienodes wordt de *ruggengraat* of *spine* van de reductie genoemd.

Bij het afdalen in de spine wordt een stack bijgehouden (de *spine stack*) die voor elke node die hij tegenkomt een pointer naar die node pusht. Wanneer de operator is gevonden, bevat de spine stack dus aan de top een pointer naar de operator zelf, aan de bodem een pointer naar de node vanwaar het algoritme begon te zoeken, en daartussenin pointers naar de applicatienodes die de twee uiteindes verbinden. De spine stack zorgt ervoor dat we snel aan de argumenten van de operator kunnen. Ik heb de spine stack geïmplementeerd aan de hand van een gelinkte lijst, aangezien deze arbirtrair groot kan worden.

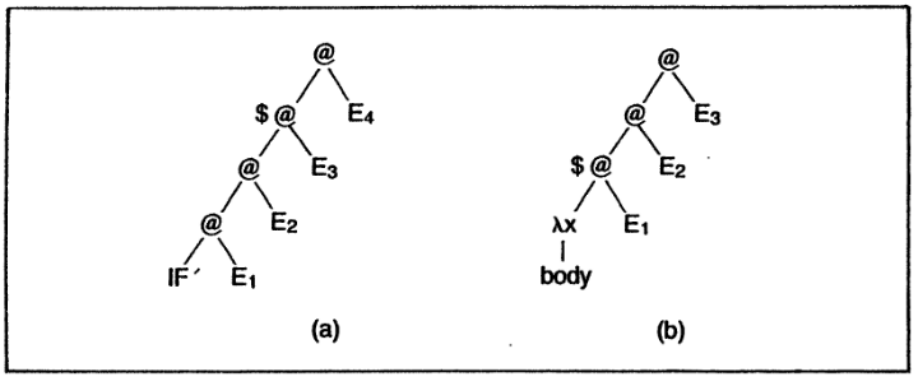
Wanneer `f` gevonden is beschouwen we volgende mogelijkheden:

1. `f` is een data-object: De expressie is in WHNF dus we zijn klaar.

Figuur 1: Illustratie van de spine stack



Figuur 2: Voorbeelden van redex selectie (redex gemarkeerd met '$')



2. `f` is een builtin operator of constructor die k argumenten vereist: De expressie is in WHNF als en slechts als $n < k$. In het andere geval is de redex de expressie `f E1 ... Ek`. De reductie die erop volgt volgt builtin-specifieke regels.

3. `f` is een λ-abstractie: Als er een argument beschikbaar is, dan is `F E1` de redex.

3.3 Graafreductie

Na het algoritme uit de voorgaande sectie uit te voeren, weten we welke node de redex is, welke de operator, en alle argumenten staan op de spine stack. Het uitvoeren van een reductie resulteert in een lokale transformatie van de graaf: de redex wordt overschreven door het resultaat van de reductie. De

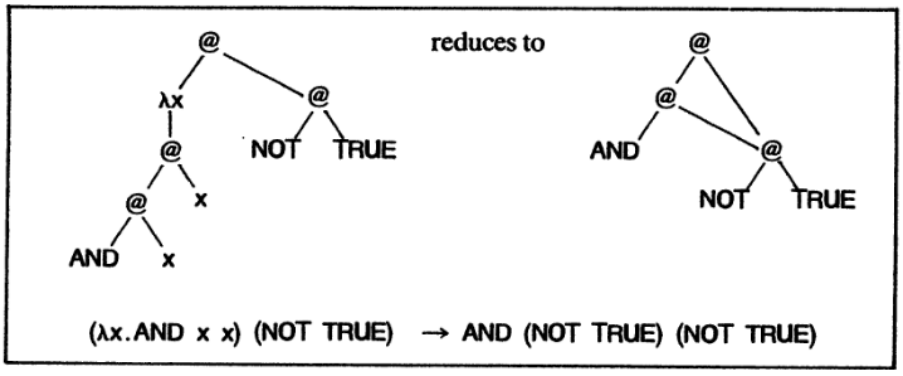
reductieprocedure wordt bepaald door het type van de operator. Ten eerste is er het triviale geval dat de operator een `DATA` cel is: in dit geval moet er niets gebeuren. *IFPL* maakt verder onderscheid tussen lambda-abstracties en builtin functies. Mijn implementatie beschouwt ook constructors nog als een apart geval.

3.3.1 Reductie van λ-abstracties

In tegenstelling tot strikte talen gaat een luie taal het argument van een functie niet eerst evalueren voordat het argument wordt gesubstitueerd voor de vrije voorvallen van de formele parameter. Een naïve aanpak zou zijn om elk voorval van de formele parameter te vervangen door een kopie van het argument. Het is makkelijk te zien dat dit tot dubbel werk kan leiden: wanneer de parameter meerdere keren wordt gebruikt in het lichaam van de functie zal mogelijks de evaluatie van de redexes in het argument meerdere keren plaatsvinden. Verder kan het ongeëvalueerde argument zelf redexes bevatten en dus arbitrair groot en complex zijn. Kopieën hiervan maken zou veel geheugen kunnen kosten.

De oplossing voor dit probleem is de reden waarom we met grafen en niet met bomen werken: we gaan elk voorval van de parameter vervangen door een *pointer* naar het argument.

Figuur 3: Pointersubstitutie

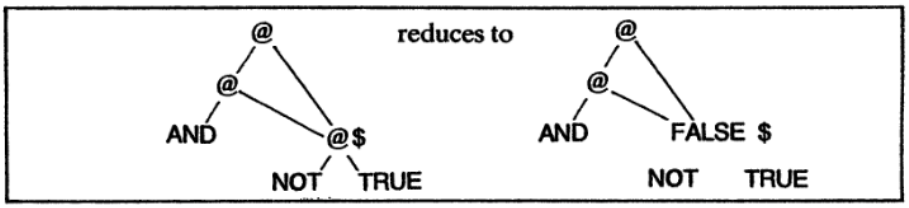


Als een argument nooit wordt gebruikt wordt het nog steeds nooit geëvalueerd, maar als het resultaat van de evaluatie van een argument minstens 1 keer nodig is, wordt het beschikbaar gemaakt voor alle voorvallen ervan.

Het wordt nu ook duidelijk waarom we de wortel van de redex na evaluatie steeds *overschrijven* met het resultaat. Deze strategie zorgt ervoor dat alle

geïnteresseerde partijen zonder meerkost een beeld krijgen op het resultaat: de node waar hun pointers naar wijzen wordt aangepast zodat het resultaat steeds onmiddellijk beschikbaar is.

Figuur 4: Overschrijven van de wortel van de redex



Het is niet onwaarschijnlijk dat een programma meerdere keren dezelfde lambda-abstracties wenst te gebruiken. Het zal dus niet mogelijk zijn om tijdens het evaluatieproces in het lichaam van de lambda-expressie zelf aanpassingen te maken. Het is dus nodig om voor elke applicatie van een lambda-expressie een nieuwe kopie te maken van het lichaam, en daarin de nodige substituties te doen. We voorzien dus een instantiate functie die in 1 pass het lambda lichaam kopieert en waar nodig substituties doet.

We hebben nu de essentie van lazy graph reduction beschreven. Het is een strategie die

1. gebruik maakt van normal-order evaluatie naar WHNF
2. die pointer substitutie gebruikt bij reductie van lambda expressies
3. die de wortel van de gereduceerde expressie steeds overschrijft met het resultaat

3.3.2 Reductie van builtin functies

Builtin functies vereisen steeds een vast aantal argumenten voordat reductie kan beginnen (`+` heeft bijvoorbeeld steeds 2 argumenten nodig), en ze gebruiken vaak ook een afwijkende reductiestrategie voor hun argumenten.

Arithmetische of logische functies vereisen bijvoorbeeld steeds dat al hun argumenten volledig gereduceerd zijn (dus geen redexes meer bevatten). Ze kunnen namelijk enkel opereren op `DATA` cellen. Het feit dat volledige reductie moet leiden tot data-objecten zou al verzekerd moeten zijn door de typechecker, dus dit wordt niet gecontroleerd op dit niveau. Een argument dat eerst volledig gereduceerd moet worden noemen we een *strikt* argument. De evaluatie van een

strikt argument gebeurt nog steeds in *normal order*, maar het reductieproces houdt niet op bij WHNF en gaat door tot normal form.

Sommige builtins hebben zeer specifieke eisen wat betreft reductieregels. Bijvoorbeeld, de `IF` builtin evalueert eerst zijn eerste argument naar normal form, dan wordt een argument geselecteerd op basis van het resultaat van het eerste argument, wat dan wordt teruggegeven zonder reductie.

3.3.3 Reductie van constructors

Mijn implementatie voorziet gestructureerde data met behulp van speciale cellen die een variabele lengte hebben. Stel dat in de high-level taal een `LIST` somtype gedefinieerd is, met een `NIL` en een `CONS a (LIST a)` constructor. We hebben dan 1 type met een 0-ary constructor en een binary constructor. De runtime omgeving moet niet kunnen onderscheiden tussen LIST types en andere types, maar wel tussen NIL en CONS dataobjecten. De typeinformatie wordt niet meegegeven aan het graafreductiealgoritme. Het is dus nodig de constructor van een type-onafhankelijke notatie te voorzien die wel informatie over de gebruikte constructor en het benodigd aantal argumenten meedraagt. Een `CONSTR` cel bevat dus 2 getallen: een *structured data tag*, een getal dat met een constructor is geassocieerd (bijvoorbeeld 0 stelt een NIL cel voor en 1 een CONS cel), en een *arity*, het nodig aantal argumenten.

Ook constructors mogen hun argumenten niet strikt evalueren. Een gestructureerde datacel moet dus eenvoudigweg verwijzingen naar haar argumenten bijhouden. Een constructie van de vorm `CONSTR-t-n a1 ... an` wordt gereduceerd naar een `DATA` cel die in haar eerste veld de structured data tag t opslaat, en in haar tweede veld een verwijzing naar een array van n `Cell`-pointers, die elk een verwijzing naar een argument opslaan. Met een andere ingebouwde functie, `SELECT`, kan men dan weer de argumenten uit de gestructureerde datacel ophalen.

Verslag 3: Typesysteem met polymorfisme en inferentie

Koen Pauwels

21 juni 2016

1 Inleiding

Een van de eigenschappen die de de familie van sterk getypeerde zuiver functionele onderscheidt is het typesysteem. De meest in het oog springende eigenschap van het typesysteem van talen zoals Haskell en ML is dat het vrijwel nooit strikt vereist is dat de programmeur de types van variabelen specifieert, terwijl de taal wel statisch getypeerd is. De reden hiervoor is dat de compiler een proces toepast dat *type inference* wordt genoemd: het afleiden van types aan de hand van de context. Een typechecker die aan type-inferentie doet is een die tegelijk kan bepalen of een programma correct getypeerd is, en indien dat het geval is, wat het type van elke expressie in het programma is.

Een andere belangrijke eigenschap is polymorfisme: de functies die we definiëren in onze taal zullen vaak geen concrete types opleggen aan bepaalde parameters of teruggegeven waarde.

Sommige functieparameters zal geen enkele beperking worden opgelegd. Een voorbeeld hiervan is de `id x = x` functie. Het is duidelijk dat de identiteitsfunctie een operatie is die in principe op een waarde van eender welk type kan worden gebruikt. `id 3`, `id ‘‘hallo’’` en `id id` zijn allemaal geldige toepassingen van id. Ons typesysteem zal dit toelaten door `id` het type `a -> a` toe te kennen, waarbij `a` een arbitraire *typevariabele* is. Zo'n typevariabele kan worden vervangen door eender welk type zolang de beperkingen opgelegd door de type-expressie in acht worden genomen. Het feit dat de `a` zowel voor als na de pijl voorkomt, dicteert bijvoorbeeld dat `id` een functietype moet hebben met dezelfde input als output.

De vorm van polymorfisme die hier wordt gebruikt wordt ook wel *parametrisch polymorfisme* genoemd. Deze term onderscheidt deze vorm van polymorfisme

van *ad-hoc polymorfisme*. Met ad-hoc polymorfisme wordt bedoeld dat een functie instanties van verschillende types kan hebben, maar dat er een expliciete implementatie voorzien is voor elke instantie. Een voorbeeld hiervan is de + operator, die in veel talen een aparte definitie heeft voor gehele getallen, reële getallen, en eventueel nog andere types. Contrasteer met het voorbeeld van de `id` functie, waar de compiler automatisch de functie voor elk benodigd type kan instantiëren. Mijn implementatie ondersteunt enkel parametrisch polymorfisme.

2 De intermediaire taal

De typechecker zal opereren op een taal die een subset van de verrijkte lambda-calculus is, maar een superset van de gewone lambda-calculus waarnaar we compileren. Deze intermediaire taal ondersteunt behalve variabelen, applicaties en lambda-abstracties ook let en letrec expressies. In de intermediaire taal voorzien we geen pattern matching.

Het kan misschien vreemd lijken dat let-expressies nodig zijn op dit niveau van de taal, aangezien een let-expressie `let x = e in e'` eenvoudig kan worden omgezet naar `(λx.e') e`. Toch is het zinvol om op het niveau van de typechecker een onderscheid te behouden tussen variabelen die geïntroduceerd werden door een let-binding enerzijds, en formele parameters van functies anderzijds. Het onderscheid berust op de vraag of verschillende voorvallen van dezelfde variabele hetzelfde type moeten hebben of niet - het antwoord op deze vraag verschilt voor lambda-gebonden variabelen versus let-gebonden variabelen.

Beschouw de functie `F = λf.λa.λb.λc.c (f a) (f b)` (voorbeeld uit *IFPL*). Het is niet onmiddellijk duidelijk of de compiler moet eisen dat de twee voorvallen van `f` hetzelfde type moeten hebben of niet. Het is zeker mogelijk om voorbeelden te bedenken van toepassingen van `F` waar de voorvallen van `f` verschillende types hebben, bijvoorbeeld `F id 0 'a'` geeft een functie terug die een functie `c` als parameter verwacht van het type `(Int -> Char -> a)`. Maar het is ook mogelijk om een voorbeeld te bedenken waarbij het misloopt. Bijvoorbeeld `F code 0 'a' K` waarbij de code functie enkel gedefinieerd is voor characters, zou resulteren in de evaluatie van de expressie `(code 0)`, wat een type error is. In acht nemend dat onze type checker niet uitsluitend moet werken in een beperkte verzameling goed uitgekozen gevallen maar in alle omstandigheden, moeten we uitgaan van het slechtste geval en moeten we dus eisen dat alle voorvallen van lambda-gebonden variabelen hetzelfde type moeten hebben. In het bovenstaande voorbeeld zorgt dat ervoor dat `a` en `b` ook van hetzelfde type moeten zijn.

Een van de eigenschappen van het typesysteem is parametrisch polymorfisme, de eigenschap dat een waarde meerdere types kan hebben naargelang de context. Het is duidelijk dat deze eigenschap niet kan bestaan als we de beperking op lambda-gebonden variabelen ook opleggen aan let-gebonden variabelen.

Bij het omzetten van de verrijkte lambda-calculus naar de gewone lambda-calculus worden letrec-expressies vertaald met behulp van een fixpunt combinator (tenminste conceptueel, de combinator kan als een builtin geïmplementeerd worden). Men kan zich dus afvragen waarom we kiezen om letrec-expressies ook nog te ondersteunen in de intermediaire taal. Want hoewel we geen Y-combinator kunnen definiëren in de high-level taal zelf (de type checker kan niet om met een oneindig type), kunnen we deze wel introduceren als een builtin met het vooraf toegewezen type `Y::(a -> a) -> a`. Dit is een mogelijkheid, maar de auteur van het boek is van mening dat dit "het probleem onder de mat vegen" is, en verkiest om recursie op een explicietere manier te behandelen.

3 De typechecker

Stel dat we de expressie `(e1 e2)` willen typechecken, waarbij het type van `e1` `t1` is, en dat van `e2` is `t2`. Dit is equivalent met het oplossen van de vergelijking `t1 = t2 -> (TypeVar n)` waarbij n een nooit eerder gebruikte naam van een typevariabele is. Als we de typering van een heel programma wensen te evalueren, zullen we een systeem van zulke vergelijkingen moeten oplossen.

Een oplossing voor zo'n systeem zal worden uitgedrukt als een functie die typevariabelen (de onbekenden van het systeem) afbeeldt op type-expressies. We drukken dit expliciet uit in Haskell met het type

```
type Subst = Symbol -> Type
```

waarbij Symbol de naam van een typevariabele is. Samengevat is het doel van de typechecker om een substitutiefunctie te vinden die een stelsel van typevergelijkingen oplost als zo'n oplossing bestaat, en een error teruggeeft wanneer zo'n oplossing niet bestaat. Substitutiefuncties zullen worden genoteerd met uitgeschreven Griekse letters, dus bijvoorbeeld `phi` of `psi`.

3.1 Substituties

Een `Subst` werkt op individuele typevariabelen; we definiëren hulpfuncties `sub_type`, die een substitutiefunctie toepast op een type-expressie, en `scomp`, die de compositie van twee substitutiefuncties teruggeeft.

```
sub_type :: Subst -> Type -> Type
scomp :: Subst -> Subst -> Subst
```

De identiteitssubstitutie toepassen op een type geeft steeds het type zelf terug. Een delta-substitutie is een substitutie die slechts 1 typevariabele beïnvloedt.

```
id_subst :: Subst
delta :: Symbol -> Type -> Subst
```

De opbouw van de uiteindelijke substitutiefunctie begint bij de identiteitssubstitutie, die variabele per variabele zal worden opgebouwd door compositie met delta-substituties. Aangezien elk type zelf typevariabelen kan bevatten (denk bijvoorbeeld een substitutie die `a` afbeeldt op `[b]` met a en b typevariabelen), is het mogelijk dat er meerdere achtereenvolgende substituties nodig zijn om een type-expressie te transformeren in een concreet type. We zullen echter zoeken naar volledig uitgewerkte substituties die slechts eenmaal moeten worden toegepast. Dit idee kan formeel worden gemaakt: we zoeken een substitutie `phi` waarvoor geldt: `(sub_type phi).(sub_type phi) = sub_type phi`, m.a.w. phi is idempotent.

3.2 Unificatie

Het proces waarmee we een substitutie construeren die een verzameling van typevergelijkingen oplost, noemen we unificatie.

```
unify :: Subst -> (Type, Type) -> Maybe Subst
```

De `unify` functie verwacht een idempotente substitutie en een paar van type-expressies. Wanneer er een uitbreiding `psi` bestaat van `phi` die een idempotent is en (t1, t2) verenigt zal de `unify` functie deze vinden en teruggeven. Het algoritme staat ons toe een substitutie die een oplossing biedt voor een systeem van k vergelijkingen uit te breiden naar een substitutie die een oplossing biedt voor een systeem van k+1 vergelijkingen.

3.3 De type environment

We associëren met elke variabele een soort template-type waarin twee soorten typevariabelen onderscheiden worden: schematische en niet-schematische. Dit template noemen we een *typeschema*. De schematische typevariabelen in een typeschema geassocieerd met een variabele zijn de typevariabelen die vrij mogen worden geinstantieerd om te conformeren met de type constraints op de

verschillende voorvallen van die variabele. De niet-schematische typevariabelen zijn constrained, en mogen niet geinstantieerd worden wanneer het typeschema geinstantieerd wordt. We noemen deze laatste ook de *onbekenden*.

```
data TypeScheme = TypeScheme [Symbol] Type
```

Hier is de lijst van Symbols de lijst van schematische variabelen. Alle andere typevariaben die in het Type voorkomen zijn onbekenden.

Zo'n typeschema zal met elke vrije variabele in een expressie worden geassocieerd. De associatielijst die variabelen aan typeschema's linkt, noemen we de *type environment*.

```
type TypeEnv = [(Symbol, TypeScheme)]
```

3.4 Het typecheck algoritme

De typechecker zal een functie zijn van de vorm `(tc gamma ns e)` met

- `gamma` is een type environment die typeschema's associeert met de vrije variabelen in e.
- `ns` is een bron van nieuwe typevariabelenamen.
- `e` is de expressie die we gaan typechecken.

```
tc :: TypeEnv -> NameSupply -> VExpr
   -> Maybe (Subst, Type)
```

Als de expressie goed getypeerd is, zal het typecheck algoritme een substitutie voor de onbekenden teruggeven, en een type voor e. Het algoritme zal uiteenvallen in een geval voor elk van de 5 constructies die ondersteund worden door de intermediaire taal.

3.4.1 Variabelen typechecken

We beginnen met het typeschema op te zoeken dat geassocieerd is met deze variabele. In het type van het typeschema kunnen schematische variabelen en onbekenden voorkomen.

We wijzen een type toe aan de variabele als volgt: de schematische variabelen worden vervangen door een nieuwe, unieke typevariabele; de onbekenden worden gelaten zoals ze zijn.

De typechecker geeft als substitutie de identiteitssubstitutie terug. Het typechecken van variabelen geeft geen informatie over onbekenden.

3.4.2 Applicaties typechecken

Eerst trachten we een substitutie `phi` te construeren die de type constraints op de twee deelexpressies samen oplost. Stel vervolgens dat `t1` en `t2` de types zijn die afgeleid zijn voor e1 en e2 (door recursief de typechecker op de deelexpressies op te roepen), dan kunnen we de typecheck afronden door een uitbreiding van `phi` te construeren die ook aan de constraint `t1 = t2 -> t'` (met t' een nieuwe typevariabele) voldoet.

Het uiteindelijke type dat aan de applicatie toegewezen zal worden, wordt gevonden door de substitutiefunctie aan het einde van deze stappen toe te passen op `t'`.

3.4.3 Lambda abstracties typechecken

We typechecken de expressie `(λ x.e)` Zoals eerder gezegd wensen we op te leggen dat elk voorval van de formele parameter x van de abstractie in het lichaam hetzelfde type opgelegd krijgt. We bekomen dit door in de type environment met de formele parameter een typeschema te associëren dat een nieuwe typevariabele introduceert die niet schematisch is. In de type environment vinden we dus de entry `(''x'', TypeScheme [] (TypeVar tvn))` terug, met tvn een nieuwe type variable name.

Deze aangepaste type environment `gamma'` wordt dan meegegeven met de recursieve oproep die het type van het lichaam van de abstractie zal bepalen.

3.4.4 Let(rec) expressies typechecken

In de expressie `let x = e' in e` achterhalen we eerst het type van `e'`. We updaten de environment met het gepaste typeschema voor x en typechecken tenslotte `e`.

Het geval voor letrec expressies is complexer.

- We beginnen met nieuwe typeschema's te verbinden aan de linkerleden van de definities. Deze schema's hebben geen schematische variabelen, aangezien we vereisen dat de voorvallen van een naam in de rechterzijde van zijn eigen definitie allemaal hetzelfde type moeten hebben.

- We typechecken de rechterzijdes, wat ons een substitutie en een lijst van types oplevert.
- We passen unify toe op de de types die afgeleid zijn voor de rechterzijdes met de overeenkomstige variabelen aan de linkerzijdes in de context van de substitutie die we bekomen hebben met de vorige stap.
- Nu zijn de definities verwerkt en is de situatie analoog aan die van een gewone let expressie. We moeten enkel nog het lichaam `e` updaten.

Dit vervolledigt de definitie van de type checker.