Verslag 2: Lazy evaluation door middel van graafreductie

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 waarin we 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 variabele worden geassocieerd. De associatielijst die variabelen aan typeschema's linkt, noemen we de *type environment*.

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

3.4 Het typecheck algoritme