

Verslag 3: Typesysteem met polymorfisme en inferentie

Koen Pauwels

21 juni 2016

1 Inleiding

Een van de eigenschappen die 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 specificeert, 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 $F = \lambda f. \lambda a. \lambda b. \lambda c. c \ (f \ a) \ (f \ b)$ (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 $Y :: (a \rightarrow a) \rightarrow a$. 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 $t1 = t2 \rightarrow (\text{TypeVar } n)$ 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  $\rightarrow$  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 ϕ of ψ .

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 identiteits-substitutie, 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: $(\text{sub_type } \text{phi}).(\text{sub_type } \text{phi}) = \text{sub_type } \text{phi}$, 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 geïnstantieerd om te conformeren met de type constraints op de

verschillende voorvallen van die variabele. De niet-schematische typevariabelen zijn constrained, en mogen niet geïnstantieerd worden wanneer het typeschema geïnstantieerd 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 typevariabelen 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.