Formalisatie Languate

Pieter Vander Vennet

June 18, 2015

1 Syntax

Zie de bnf-bestanden voor de volledige syntax met syntactische suiker.
Hieronder volgt er een kort, involledig overzicht van de belangerijkste regels.

1.1 Expressies

```
functionName     ::= [a..z][a..zA..Z0..9]
constructorName  ::= [A..Z][a..zA..Z0..9]
-- een expressie is een functieoproep met argumenten
-- een functionName kan ook een locale variabele zijn
expression ::= (functionName | constructor) expression*
               | "(" expression ")"
```

Een constructor is bv. **True**, **False**, maar ook **Nil** en **Elem**.
Een expressie is bv **Elem True (Elem False Nil)**

1.2 Types

```
knownType       ::= [A..Z][a..zA..Z0..9]
freeType        ::= [a..z][a..zA..Z0..9]
constrainedType ::= "("
simpleType      ::= knownType | freeType | "(" type ")"
appliedType     ::= simpleType+
type            ::= appliedType ("->" type) *
```

Types zijn bv. **Bool**, **Bool →Bool**, **(a →b) →List a →List b**

1.3 Nieuwe types

```
data    ::= "data" knownType freeType* "="
          (constructor type*) ("|" constructor type*)*
cat             ::= "cat" knownType freeTypes
                        ("\n" functionDeclaration)+
instance        ::= "instance" type "is" type
```

Voorbeelden hiervan zijn:

```
data Bool       = True | False
data List a     = Nil | Elem a (List a)

cat Functor a
    map : (a -> b) -> (functor:Functor) a -> functor b

instance List is Functor
```

1.4 Functiedeclaraties en pattern matching

```
variable::= functionName
pattern ::= variable | constructor | "(" pattern+ ")"
function::= functionName ":" type ("&" type)* "\n"
        (pattern "=" expression)+
```

Examples of functions are:

```
not      : Bool -> Bool
True     = False
False    = True

and              : Bool -> Bool -> Bool
True True        = True
_ _              = False

map              : (a -> b) -> List a -> List b
_ Nil            = Nil
f (Elem a list) = Elem (f a) (map f list)
```

2 Typeringsregels

Ξ is de typeringscontext en houdt bij welke functie welk type heeft op een globaal niveau. Expressies worden geëvalueerd binnen een omgeving ξ, welke de types (en waardes) van lokale variabelen bijhoudt. Γ houdt de subtyperingsrelatie bij. θ weet welke clause van welk type is. Wanneer deze tabellen niet nodig zijn, worden ze niet vermeld in de typeringsregels. De grijze text stelt een ingegeven lijn code voor, **syntax in het vet** duiden op variabele namen.

2.1 Typering van constructoren

$$\frac{\Xi,\ \text{data } \mathbf{T} = \mathbf{Cons}}{\Xi \cup \mathbf{Cons} : \mathbf{T}} \text{ (Simpele constructor)}$$

$$\frac{\Xi,\ \text{data } \mathbf{T} = \mathbf{Cons\ T0\ T1\ \ldots}}{\Xi \cup \mathbf{Cons} : \mathbf{T0} \to \mathbf{T1} \to \ldots \to \mathbf{T}} \text{ (Constructor met type argumenten)}$$

$$\frac{\Xi,\ \text{data } \mathbf{T} = \mathbf{Cons0\ \ldots}\ |\ \text{Cons1} \ldots |\ \ldots}{\Xi \cup \mathbf{Cons0} : \ldots \to \mathbf{T} \cup \mathbf{Const1} : \ldots \to \mathbf{T} \cup \ldots} \text{ (Data met meerdere constructoren)}$$

2.2 Typering van patterns

Patterns halen een waarde uit elkaar, en slaan die op in een lokale variabele, binnen de tabel ξ (lokale context). Patterns worden getypeerd met behulp van een type, hier altijd T genoemd.

$$\frac{\xi,\ \mathbf{a}\ ,\ \mathrm{T}}{\xi \cup \mathbf{a} : \mathrm{T}} \text{ (Variabele-toekenning)}$$

$$\frac{\xi,\ \Gamma \vdash \mathbf{Cons} \in \mathbf{T},\ \text{Cons}\ ,\ \mathrm{T}}{\xi \cup \text{Cons} \in \mathrm{T}} \text{ (Simpele Patternmatch)}$$

$$\frac{\xi \vdash \mathrm{P0} \in \mathrm{T0},\ \mathrm{P1} \in \mathrm{T1},\ \ldots,\ \Gamma \vdash \mathbf{Cons} \in \mathrm{T0} \to \mathrm{T1} \to \ldots \to \mathrm{T},\ \mathbf{Cons\ P0\ P1\ \ldots}\ ,\ \mathrm{T}}{\xi \cup \mathbf{Cons\ P0\ P1} \ldots \in \mathrm{T}} \text{ (Recursieve patroontypering)}$$

2.3 Typering van clauses

We voeren nog een tabel in: θ. Deze is een 'lokale variabele' die bijhoudt welke expressie van welk type is.

De eerste regel ziet er vreemd uit, dit is omdat er geen patterns zijn.

$$\frac{\theta, \Xi \vdash \mathbf{expr} : \mathrm{T},\ \ \mathbf{= expr}}{\theta \cup\ \ \mathbf{= expr} : \mathrm{T}} \text{ (No patterns-clause)}$$

$$\frac{\theta \vdash \mathbf{P1\ \ldots = expr} : \mathrm{T0} \to \mathrm{T},\ \Xi, \xi \vdash \mathbf{P0} : \mathrm{T0},\ \mathbf{P0\ P1\ \ldots = expr}}{\theta \cup \mathrm{P0\ P1\ \ldots} = \mathbf{expr} : \mathrm{T}} \text{ (Recursive-pattern application)}$$

2.4 Typering van functies

$$\frac{\begin{array}{l}\Gamma, \Xi, \xi, \theta \vdash \text{clause0} : \mathrm{T0},\ \mathrm{T1},\ \ldots,\ \text{clause1} : \mathrm{T0},\ \mathrm{T1},\ \ldots \\ \quad \text{f} : \mathrm{T0}\ \&\mathrm{T1}\ \&\ldots \backslash\mathrm{n}\ \text{clause0}\ \backslash\mathrm{n}\ \text{clause1}\ \ldots\end{array}}{\Gamma \cup \mathrm{f} : \mathrm{T0}\ \&\mathrm{T1}\ \&\ldots} \text{ (functie-declaratie)}$$

Technisch gezien is ξ en θ verschillend per clause. Dit is immer de local scope die bij de bijbehorende pattern match hoort. Elke clause moet een supertype zijn van f. Elke clause moet dus een supertype zijn van $T0$,$T1$, ...

2.5 Subtyperingsregels

Om de regels simpel te houden, houden we geen rekening met de vrije type variabelen. Deze kunnen immers hernoemd worden wanneer nodig, we gaan ervan uit dat dit impliciet gebeurt.

$$\frac{\Gamma \vdash \mathrm{e} : \mathrm{T},\ \text{instance } \mathbf{T} \text{ is } \mathbf{SuperT}}{\mathrm{e} : \mathrm{SuperT}} \text{ (instantie-declaratie)}$$

3 Evaluatieregels

3.1 Expressies

Een expressie wordt geëvalueerd binnen een context ξ naar een *waarde*. Deze waarde is enerzijds een *Constructor*, of anderzijds een *Thunk*, die de resterende expressie en context bijhoudt. Op deze manier kan lazyness en partiële applicatie voorgesteld worden.

$$\frac{\xi \vdash \text{f} : \text{T},\ \text{f}}{\text{Thunk } [\xi 0 \text{ clause0}, \xi 1 \text{ clause1}, \ldots]} \text{ (Wrapping in context)}$$

$$\frac{\text{Thunk } [\text{clause0}, \text{clause1}, \ldots] \quad \text{val}}{\text{Thunk}[\text{clause0 val}, \text{clause1}, \ldots]} \text{ (Applicatie)}$$

$$\frac{\text{Thunk } [\xi 0 = \textbf{expr} \quad \textbf{a} \quad \textbf{b} \ , \ldots]}{\Gamma,\ \text{expr} \quad \text{a} \quad \text{b}} \text{ (Selectie eerste matchende clause)}$$

De eerste clause waar alle patterns matchen wordt geselecteerd. De variabelen in de context ξ worden gesubstitueerd in de overblijvende expressie. Aangezien ook functies variabelen zijn, worden deze allemaal gesubstitueerd tot values, waarna deze met applicatie of andere regels verder geëvalueerd worden.

$$\frac{\xi \vdash \text{P0} : \text{T} \ , \text{val} : \text{T}, (\text{P0 P1} \ldots = \text{expr}) \ \text{val}}{\xi \cup \textit{P0\%val}, \text{P1} \ldots = \text{expr}} \text{ (Pattern matching)}$$

Een constructor wordt geëvalueerd tot een ADT waarbij de index van de constructor wordt gebruikt.

$$\frac{\text{Thunk } \xi, \textbf{Constructor val0 val1 ...}}{\text{ADT i val0 val1} \ldots} \text{ (Constructor evaluatie)}$$

3.2 Patterns

Patterns worden uit elkaar getrokken en geven aanleiding tot een lokale scope.

$$\frac{\xi,\ \text{a} \ \% \ \text{val}}{\text{a} = \text{val}} \text{ (Variabele-toekenning)}$$

$$\frac{\xi,\ (\text{Constructor i P0 P1} \ldots) \ \% \ \text{ADT i val0 va1} \ldots}{\text{P0} \ \% \ \text{val0} \ \cup \ \text{P1} \ \% \ \text{val1} \ \cup \ldots} \text{ (Deconstructie)}$$