

Functioneel programmeren met Helium



Bastiaan Heeren en Daan Leijen
{bastiaan,daan}@cs.uu.nl
Universiteit Utrecht

NIOC 2004, Groningen, 3 november.

- ▶ Functioneel programmeren
 - Haskell
 - Type inferentie
 - Nieuwe datatypes
 - Haskell in het onderwijs

- ▶ De Helium Compiler
 - Foutmeldingen en waarschuwingen
 - Registratiesysteem
 - Type inferentie directieven
 - wxHaskell

Functioneel Programmeren

- ▶ *Imperatieve* programmeertalen
 - bijvoorbeeld: C, Java, Pascal, en Ada
 - een reeks commando's, uitgevoerd in een stricte volgorde
- ▶ *Functionele* programmeertalen
 - bijvoorbeeld: Haskell, Standard ML, Caml, Clean, en Erlang
 - één expressie, welke wordt geëvalueerd
 - andere voorbeelden: spreadsheets en SQL
- ▶ Het grote verschil: *wat* in plaats van *hoe*
- ▶ Volgorde van evaluatie is onbekend, dus geen assignment

Haskell

- ▶ Een polymorf getypeerde, lazy, puur functionele programmeertaal
- ▶ Vernoemd naar Haskell Brooks Curry (wiskundige logicus)
- ▶ Gebaseerd op de lambda calculus



- ▶ Voordelen:

- Hogere productiviteit (factor 9-25, Ericsson)
- Code is korter, duidelijker, en beter onderhoudbaar (factor 2-10)
- Minder fouten, grotere betrouwbaarheid (geen core dumps)
- Zeer geschikt voor domein specifieke problemen
- Kortere *lead times*

- ▶ Nadelen:

- Minder snel dan bijvoorbeeld C
- Vereist een ander perspectief op programmeren

Faculteitsfunctie in Java (iteratief)

```
class Fac
{
    public static void main(String ps[])
    {
        System.out.println(fac(10));
    }
    public static int fac (int n)
    {
        int total = 1;
        for (int i=1; i<=n; i++)
        {
            total = total * i;
        }
        return total;
    }
}
```

Faculteitsfunctie in Haskell

- ▶ Een recursieve definitie:

```
fac n = if n==0 then 1 else n * fac (n-1)
```

Faculteitsfunctie in Haskell

- ▶ Een recursieve definitie:

```
fac n = if n==0 then 1 else n * fac (n-1)
```

- ▶ ... of een definitie met patronen:

```
fac 0 = 1  
fac n = n * fac (n-1)
```

Faculteitsfunctie in Haskell

- ▶ Een recursieve definitie:

```
fac n = if n==0 then 1 else n * fac (n-1)
```

- ▶ ... of een definitie met patronen:

```
fac 0 = 1  
fac n = n * fac (n-1)
```

- ▶ ... of gebruik standaard functies:

```
fac n = product [1..n]
```

Faculteitsfunctie in Haskell

- ▶ Een recursieve definitie:

```
fac n = if n==0 then 1 else n * fac (n-1)
```

- ▶ ... of een definitie met patronen:

```
fac 0 = 1  
fac n = n * fac (n-1)
```

- ▶ ... of gebruik standaard functies:

```
fac n = product [1..n]
```

- ▶ ... of met een *hogere orde* functie:

```
fac n = foldr (*) 1 [1..n]
```

Faculteitsfunctie in Haskell

- ▶ Een recursieve definitie:

```
fac n = if n==0 then 1 else n * fac (n-1)
```

- ▶ ... of een definitie met patronen:

```
fac 0 = 1  
fac n = n * fac (n-1)
```

- ▶ ... of gebruik standaard functies:

```
fac n = product [1..n]
```

- ▶ ... of met een *hogere orde* functie:

```
fac n = foldr (*) 1 [1..n]
```

- ▶ ... of voor de echte hacker:

```
fac = foldr (*) 1 . enumFromTo 1
```

Redeneren over expressies

```
kwadraat x = x * x
```

- ▶ Omdat er geen side-effects kunnen optreden, kunnen we redeneren over expressies die gebruik maken van de functie `kwadraat`.

```
kwadraat (4 + 4) = kwadraat 8  
                  = 8 * 8  
                  = 64
```

Redeneren over expressies

```
kwadraat x = x * x
```

- ▶ Omdat er geen side-effects kunnen optreden, kunnen we redeneren over expressies die gebruik maken van de functie `kwadraat`.

```
kwadraat (4 + 4) = kwadraat 8
                  = 8 * 8
                  = 64
```

- ▶ Maar ook:

```
kwadraat (4 + 4) = (4 + 4) * (4 + 4)
                  = 8 * (4 + 4)
                  = 8 * 8
                  = 64
```

- ▶ Door het ontbreken van side-effects krijgen we referentiële transparantie. Hiermee kunnen we gelijkheid tussen expressies aantonen.
- ▶ De waarde van een expressie wordt alleen uitgerekend als deze nodig is (*lazy*).

Type inferentie

- ▶ Haskell is sterk getypeerd: aan elke expressie wordt een type toegekend (tijdens het compileren), zonder dat er type annotaties nodig zijn.

```
4           :: Int
(+)        :: Int -> Int -> Int
product    :: [Int] -> Int
kwadraat   :: Int -> Int
```

Type inferentie

- ▶ Haskell is sterk getypeerd: aan elke expressie wordt een type toegekend (tijdens het compileren), zonder dat er type annotaties nodig zijn.

```
4           :: Int
(+)         :: Int -> Int -> Int
product     :: [Int] -> Int
kwadraat    :: Int -> Int
```

- ▶ En dus ook:

```
kwadraat (4 + 4) :: Int
```

- ▶ Type inferentie voorkomt vele programmeerfouten.
- ▶ Milner (1978): “Well-typed programs cannot go wrong”
- ▶ Maar wat is dan het type voor bijvoorbeeld functie compositie?

Polymorfie

$$(f \ . \ g) \ x = f \ (g \ x)$$

► Oplossing 1:

$$(\cdot) \ :: \ (Int \ \rightarrow \ Int) \ \rightarrow \ (Int \ \rightarrow \ Int) \ \rightarrow \ (Int \ \rightarrow \ Int)$$

Dit is geen goed idee, want:

$$f = \text{kwadraat} \ . \ (+1)$$

ok

$$g = (>10) \ . \ \text{kwadraat}$$

fout?

Polymorfie

$$(f \ . \ g) \ x = f \ (g \ x)$$

► Oplossing 1:

$$(\cdot) \ :: \ (\text{Int} \ \rightarrow \ \text{Int}) \ \rightarrow \ (\text{Int} \ \rightarrow \ \text{Int}) \ \rightarrow \ (\text{Int} \ \rightarrow \ \text{Int})$$

Dit is geen goed idee, want:

$$f = \text{kwadraat} \ . \ (+1)$$

ok

$$g = (>10) \ . \ \text{kwadraat}$$

fout?

► Oplossing 2: subtype polymorfie

$$(\cdot) \ :: \ (\text{Obj} \ \rightarrow \ \text{Obj}) \ \rightarrow \ (\text{Obj} \ \rightarrow \ \text{Obj}) \ \rightarrow \ (\text{Obj} \ \rightarrow \ \text{Obj})$$

Werkt in dit geval ook niet, want:

$$g = (>10) \ . \ \text{kwadraat}$$

ok

$$h = \text{kwadraat} \ . \ (>10)$$

ok?

Polymorfie

$$(f \ . \ g) \ x = f \ (g \ x)$$

► Oplossing 3: parametrische polymorfie

$$(\cdot) \ :: \ (b \ \rightarrow \ c) \ \rightarrow \ (a \ \rightarrow \ b) \ \rightarrow \ (a \ \rightarrow \ c)$$

Polymorfie

```
(f . g) x = f (g x)
```

- ▶ Oplossing 3: parametrische polymorfie

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

- ▶ Nu kunnen we ook over types van expressies redeneren:

```
(.)      :: (b -> c ) -> (a -> b ) -> (a -> c)  
(>10)   :: (Int -> Bool)  
kwadraat :: (Int -> Int)
```

En dus geldt ook dat:

```
(>10) . kwadraat :: Int -> Bool
```

Nieuwe datatypes introduceren

► Een boom is...

- een blad met een expressie van het type `Int`
- of een tak met twee deelbomen

```
data IntBoom = Blad Int
              | Tak IntBoom IntBoom
```

De types van de twee *constructoren* zijn:

```
Blad  :: Int -> IntBoom
Tak   :: IntBoom -> IntBoom -> IntBoom
```

Nieuwe datatypes introduceren

► Een boom is...

- een blad met een expressie van het type `Int`
- of een tak met twee deelbomen

```
data IntBoom = Blad Int
              | Tak IntBoom IntBoom
```

De types van de twee *constructoren* zijn:

```
Blad :: Int -> IntBoom
Tak  :: IntBoom -> IntBoom -> IntBoom
```

► Een voorbeeld:

```
boompje = Tak (Blad (3 + 4))
           (Tak (Blad (5 * 5)) (Blad 6))
```

Een algemenere oplossing

```
data Boom a = Blad a
            | Tak (Boom a) (Boom a)
```

► De types van de twee *constructoren* zijn nu polymorf:

```
Blad :: a -> Boom a
```

```
Tak  :: Boom a -> Boom a -> Boom a
```

Een algemenere oplossing

```
data Boom a = Blad a
             | Tak (Boom a) (Boom a)
```

- ▶ De types van de twee *constructoren* zijn nu polymorf:

```
Blad :: a -> Boom a
```

```
Tak  :: Boom a -> Boom a -> Boom a
```

- ▶ Natuurlijk kunnen we (inductieve) functies definiëren over bomen:

```
som (Blad n)      = n
```

```
som (Tak b1 b2)  = som b1 + som b2
```

```
diepte (Blad n)   = 0
```

```
diepte (Tak b1 b2) = 1 + max (diepte b1) (diepte b2)
```

Een algemenere oplossing

```
data Boom a = Blad a
             | Tak (Boom a) (Boom a)
```

- ▶ De types van de twee *constructoren* zijn nu polymorf:

```
Blad :: a -> Boom a
```

```
Tak  :: Boom a -> Boom a -> Boom a
```

- ▶ Natuurlijk kunnen we (inductieve) functies definiëren over bomen:

```
som (Blad n)      = n
```

```
som (Tak b1 b2) = som b1 + som b2
```

```
diepte (Blad n)   = 0
```

```
diepte (Tak b1 b2) = 1 + max (diepte b1) (diepte b2)
```

```
som      :: Boom Int -> Int
```

```
diepte   :: Boom a   -> Int
```

Haskell in het onderwijs

- ▶ Haskell is geschikt voor het onderwijs, want:
 - Elegante syntax, die lijkt op standaard wiskunde.
 - De nadruk ligt op *wat*, en minder op *hoe*.
 - De taal is puur, dus kunnen we equationeel redeneren over expressies.
 - Vele fouten worden vroegtijdig afgevangen.
 - Het type van een expressie kan automatisch worden afgeleid.
 - Polymorfie is een natuurlijk concept om *algemene* functies te beschrijven, wat bovendien het hergebruik van code bevordert.
 - Nieuwe algebraïsche datatypes kunnen eenvoudig worden toegevoegd.

Haskell in het onderwijs

- ▶ Haskell is geschikt voor het onderwijs, want:
 - Elegante syntax, die lijkt op standaard wiskunde.
 - De nadruk ligt op *wat*, en minder op *hoe*.
 - De taal is puur, dus kunnen we equationeel redeneren over expressies.
 - Vele fouten worden vroegtijdig afgevangen.
 - Het type van een expressie kan automatisch worden afgeleid.
 - Polymorfie is een natuurlijk concept om *algemene* functies te beschrijven, wat bovendien het hergebruik van code bevordert.
 - Nieuwe algebraïsche datatypes kunnen eenvoudig worden toegevoegd.
- ▶ Helaas rapporteren Haskell compilers vaak cryptische foutmeldingen. Voor veel studenten is dit een serieuze belemmering om de taal (en de achterliggende concepten) te doorgronden.

Cryptische foutmeldingen

- ▶ Een kleine denkfout kan leiden tot een lastig te begrijpen foutmelding.

```
main = xs : [4, 5, 6]
      where len = length xs
            xs  = [1, 2, 3]
```

- ▶ De Hugs interpreter rapporteert het volgende:

```
ERROR "Main.hs":1 - Unresolved top-level overloading
*** Binding           : main
*** Outstanding context : (Num [b], Num b)
```

- ▶ Het is volstrekt onduidelijk hoe de fout opgelost kan worden. De melding biedt weinig ondersteuning.

Cryptische foutmeldingen

- ▶ Maar de foutmeldingen kunnen nog onduidelijker:

```
ERROR "BigTypeError.hs":6 - Type error in application
*** Expression      : sem_Expr_Lam <$ pKey "\\\" <*> pFoldr1 (sem_LamIds_Cons,sem_Lam
Ids_Nil) pVarid <*> pKey "->"
*** Term           : sem_Expr_Lam <$ pKey "\\\" <*> pFoldr1 (sem_LamIds_Cons,sem_Lam
Ids_Nil) pVarid
*** Type          : [Token] -> [((Type -> Int -> [[Char],(Type,Int,Int)]) -> Int
-> Int -> [(Int,(Bool,Int)]) -> (PP_Doc,Type,a,b,[c] -> [Level],[S] -> [S])) -> Type
-> d -> [[Char],(Type,Int,Int)] -> Int -> Int -> e -> (PP_Doc,Type,a,b,f -> f,[S]
-> [S]),[Token]])
*** Does not match : [Token] -> [[Char] -> Type -> d -> [[Char],(Type,Int,Int)]
-> Int -> Int -> e -> (PP_Doc,Type,a,b,f -> f,[S] -> [S]),[Token]]
```

- ▶ Helaas zijn dit soort foutmeldingen geen uitzondering.

De Helium Compiler

- ▶ In korte tijd ontwikkeld aan de Universiteit Utrecht.
- ▶ Doel: het rapporteren van precieze en gebruikersvriendelijke foutmeldingen om zo het onderwijs in Haskell te ondersteunen.
- ▶ Driemaal ingezet tijdens de introductiecursus Functioneel Programmeren.

De Helium Compiler

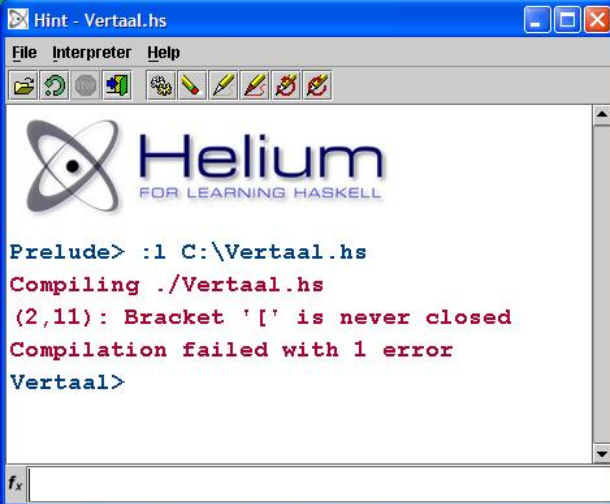
- ▶ In korte tijd ontwikkeld aan de Universiteit Utrecht.
- ▶ Doel: het rapporteren van precieze en gebruikersvriendelijke foutmeldingen om zo het onderwijs in Haskell te ondersteunen.
- ▶ Driemaal ingezet tijdens de introductiecursus Functioneel Programmeren.

Onze aanpak:

- ▶ Het inperken van de taal:
 - Helium ondersteunt bijna volledig (standaard) Haskell.
 - Het opvallendste verschil is het ontbreken van typeklassen.
- ▶ Het ontwerp is afgestemd op het geven van goede foutmeldingen.
- ▶ Helium maakt gebruik van een geavanceerd type systeem dat gebaseerd is op het verzamelen en oplossen van constraints.
- ▶ Anticiperen op veelgemaakte fouten.

Grafische interpreter

```
vertaal :: [(Int, String)]  
vertaal = [(1, "een"), (2, "twee"), (3, "drie")]
```



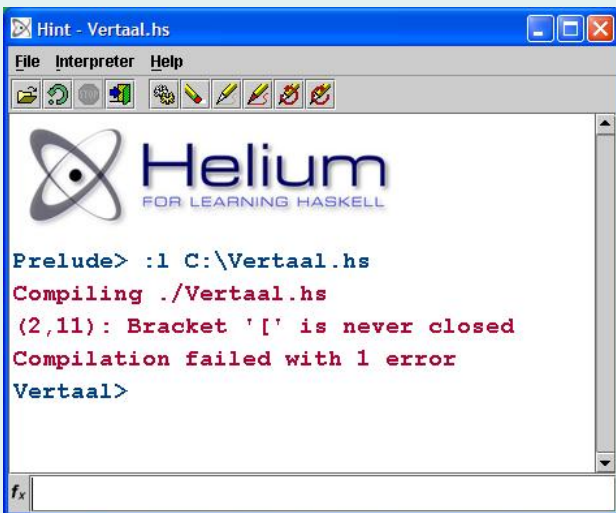
The screenshot shows a window titled "Hint - Vertaal.hs" with a menu bar containing "File", "Interpreter", and "Help". Below the menu is a toolbar with various icons. The main area displays the Helium logo (an atom symbol) and the text "Helium FOR LEARNING HASKELL". The command prompt shows the following interaction:

```
Prelude> :l C:\Vertaal.hs  
Compiling ./Vertaal.hs  
(2,11): Bracket '[' is never closed  
Compilation failed with 1 error  
Vertaal>
```

- ▶ Simpele, maar effectieve, grafische interpreter
- ▶ Exacte positie informatie (voor koppeling met editor)

Grafische interpreter

```
vertaal :: [(Int, String)]  
vertaal = [(1, "een"), (2, "twee"), (3, "drie")]
```



hugs

```
ERROR "Vertaal.hs":3 - Syntax error in expression  
(unexpected '}', possibly due to bad layout)
```

ghci

```
Vertaal.hs:3: parse error  
(possibly incorrect indentation)
```

- ▶ Simpele, maar effectieve, grafische interpreter
- ▶ Exacte positie informatie (voor koppeling met editor)

Foutmeldingen

- ▶ Ongedefinieerde variabelen - kortste edit-afstand algoritme

```
maxLengte :: [String] -> Int
maxLengte = maximun (map length xs)
```

```
(2,13): Undefined variable "maximun"
```

```
Hint: Did you mean "maximum" ?
```

```
(2,33): Undefined variable "xs"
```

```
Compilation failed with 2 errors
```

Foutmeldingen

- ▶ Ongedefinieerde variabelen - kortste edit-afstand algoritme

```
maxLengte :: [String] -> Int
maxLengte = maximun (map length xs)
```

```
(2,13): Undefined variable "maximun"
      Hint: Did you mean "maximum" ?
(2,33): Undefined variable "xs"
Compilation failed with 2 errors
```

- ▶ Dubbele definities - meerdere locaties worden vermeld

```
f x = ...
...
f x y z = ...
```

```
(1,1), (32,1): Duplicated definition "f"
Compilation failed with 1 error
```

Waarschuwingen

- ▶ Functie zonder type signatuur

$$(f \ . \ g) \ x = f \ (g \ x)$$

(2,4): Warning: Missing type signature: `(.) :: (a -> b) -> (c -> a) -> c -> b`

Waarschuwingen

► Functie zonder type signatuur

```
(f . g) x = f (g x)
```

```
(2,4): Warning: Missing type signature: (.) :: (a -> b) -> (c -> a) -> c -> b
```

► Verdachte functie compositie

```
test = sin .2
```

```
(1,12): Warning: Function composition (.) immediately followed by number
```

```
Hint: If a Float was meant, write "0.2"
```

```
Otherwise, insert a space for readability
```

Waarschuwingen

▶ Functie zonder type signatuur

```
(f . g) x = f (g x)
```

```
(2,4): Warning: Missing type signature: (.) :: (a -> b) -> (c -> a) -> c -> b
```

▶ Verdachte functie compositie

```
test = sin .2
```

```
(1,12): Warning: Function composition (.) immediately followed by number
```

```
Hint: If a Float was meant, write "0.2"
```

```
Otherwise, insert a space for readability
```

▶ Verdachte functie definities

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p [] = []
myFilter p (x:xs) =
  if p x then x : myFilter p xs else myFilter p xs
```

```
(2,1), (3,1): Warning: Suspicious adjacent functions "myFilter" and "myFilter"
```

Waarschuwingen

- ▶ Functie zonder type signatuur

```
(f . g) x = f (g x)
```

```
(2,4): Warning: Missing type signature: (.) :: (a -> b) -> (c -> a) -> c -> b
```

- ▶ Verdachte functie compositie

```
test = sin .2
```

```
(1,12): Warning: Function composition (.) immediately followed by number
```

```
Hint: If a Float was meant, write "0.2"
```

```
Otherwise, insert a space for readability
```

- ▶ Verdachte functie definities

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p [] = []
myFilter p (x:xs) =
  if p x then x : myFilter p xs else myFilter p xs
```

```
(2,1), (3,1): Warning: Suspicious adjacent functions "myFilter" and "myFilter"
```

- ▶ En verder: tab karakter, onvolledige patronen, ongebruikte variabelen, etc.

Typeringsfouten

- ▶ Type inferentie proces is geformuleerd als een constraint probleem
- ▶ De meest waarschijnlijke fout wordt gerapporteerd
 - Bij twee tegenstrijdige types wordt het aantal locaties vergeleken dat één van beide types ondersteunt.
 - Speciale heuristieken worden ingezet om veelgemaakte fouten te detecteren

```
maakEven :: Int -> Int
maakEven x = if even x then True else x+1
```

```
(2,29): Type error in then branch of conditional
expression      : if even x then True else x + 1
constructor     : True
type            : Bool
does not match : Int
```

Meer heuristieken

► Omdraaien van argumenten

```
(2,9): Type error in application
expression      : map [1 .. 10] even
term            : map
  type          : (a -> b) -> [a]          -> [b  ]
  does not match : [Int]      -> (Int -> Bool) -> [Bool]
probable fix    : re-order arguments
```

```
lijst :: [Bool]
lijst = map [1..10] even
```

Meer heuristieken

► Omdraaien van argumenten

```
(2,9): Type error in application
expression      : map [1 .. 10] even
term            : map
  type          : (a -> b) -> [a]          -> [b ]
  does not match : [Int]    -> (Int -> Bool) -> [Bool]
probable fix    : re-order arguments
```

```
lijst :: [Bool]
lijst = map [1..10] even
```

► Verwisselen van functies

```
(2,11): Type error in constructor
expression      : :
  type          : a    -> [a ] -> [a ]
  expected type : [Int] -> [Int] -> [Int]
probable fix    : use ++ instead
```

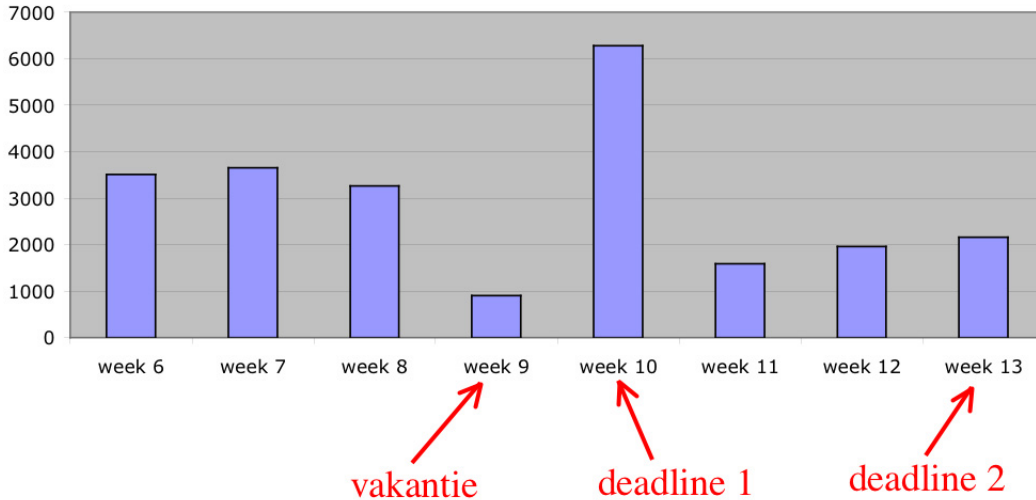
```
ints :: [Int]
ints = xs : [4, 5, 6]
  where xs = [1, 2, 3]
```

```
(:)  :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
```

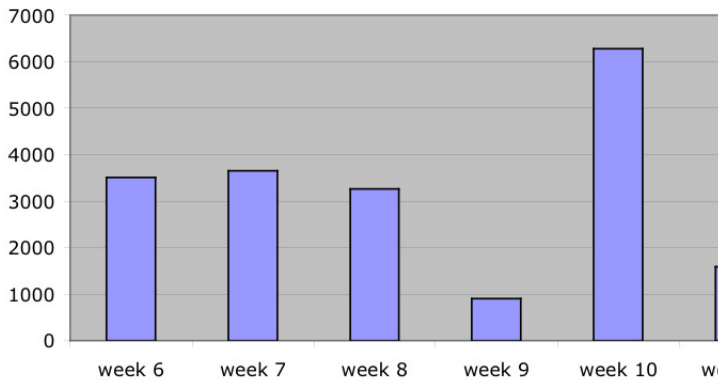
Big brother is watching

- ▶ Tijdens de introductiecursussen FP in Utrecht hebben we een registratiesysteem ingezet om studentenprogramma's te loggen.
 - Voor het krijgen van inzicht in het programmeergedrag.
 - Ter verbetering van de compiler.
- ▶ Wat wordt er geregistreerd?
 - Ieder programma dat gecompileerd wordt vanaf de studentenmachines.
 - Niet geregistreerd: expressies vanuit de interpreter.
 - Historische informatie: naam, tijdstip en versie.
- ▶ Dit heeft geleid tot een database van ruim 60.000 programma's (zowel correcte als incorrecte)

Ontvangen programma's



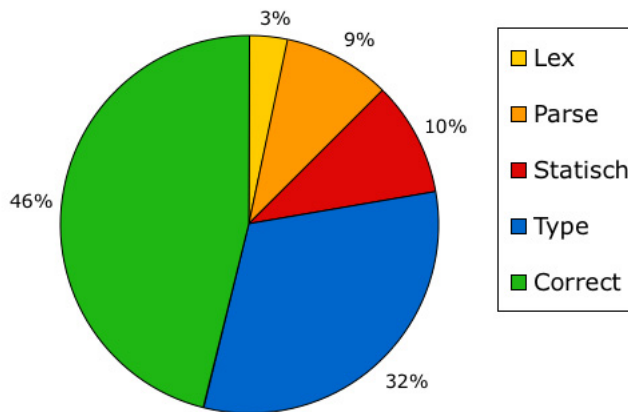
Ontvangen programma's



vakantie

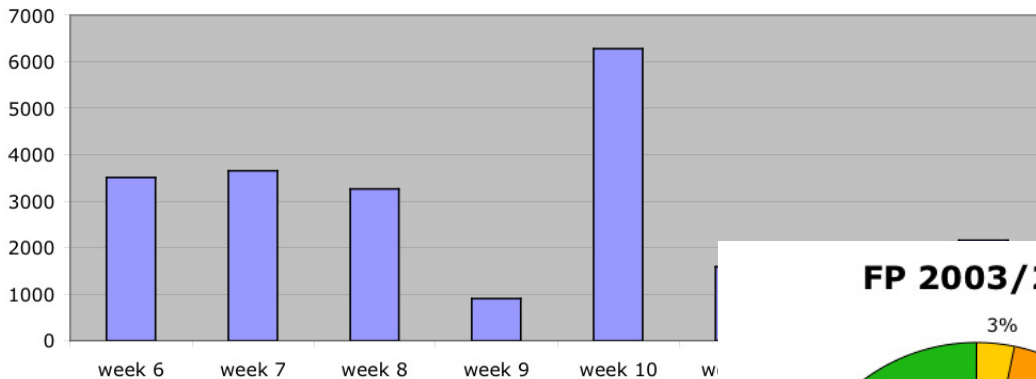
deadline

FP 2003/2004



Statistieken 03/04

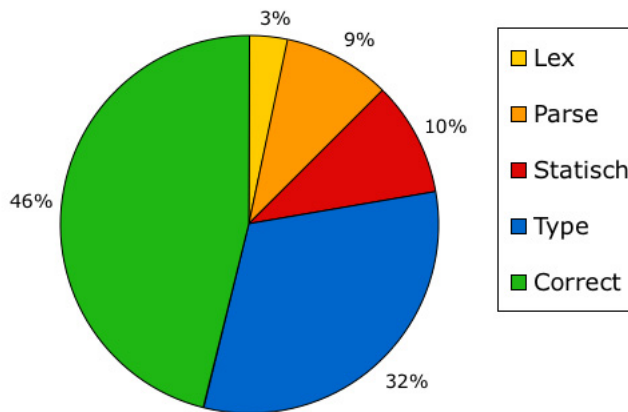
Ontvangen programma's



vakantie

deadline

FP 2003/2004



Statische fouten

Ongedefinieerde variabele	2240	55.46%
Ongedefinieerde constructor	377	9.33%
Type signatuur zonder functie	368	9.11%
Ariteit bij constructor	223	5.52%
Ariteit bij functie definitie	209	5.17%
Ongedefinieerde type constructor	196	4.85%
Dubbele definitie	157	3.89%

Type inferentie directieven

- ▶ Helium ondersteunt type inferentie directieven om van buitenaf het type inferentie proces te beïnvloeden (en in het bijzonder het rapporteren van foutmeldingen).
- ▶ Deze directieven worden opgeschreven in een *.type* bestand, welke wordt beschouwd vlak voor het type inferentie proces van het bijbehorende Haskell bestand.
- ▶ Bijvoorbeeld: declareer dat twee functies semantisch gerelateerd zijn aan elkaar. Probeer bij een typeringsfout of verwisseling het probleem verhelpt.

.type file

```
siblings    : , ++  
siblings    curry , uncurry
```

- ▶ Met speciale *typeklassen directieven* hopen we in de toekomst overloading te ondersteunen zonder kwaliteit in te leveren.

Gespecialiseerde typeringsregels

.type file

```
x :: t1;    y :: t2;
```

```
-----  
x <$> y :: t3;
```

```
t1 == a1 -> b1      : left operand is not a function
```

```
t2 == Parser s1 a2  : right operand is not a parser
```

```
t3 == Parser s2 b2  : result type is not a parser
```

```
s1 == s2 : parser has an incorrect symbol type
```

```
a1 == a2 : function cannot be applied to parser's result
```

```
b1 == b2 : parser has an incorrect result type
```

- ▶ Specialisatie van het type inferentie proces voor de <\$> combinator
- ▶ Aangepaste foutmeldingen (domein-specifiek)
- ▶ De correctheid van deze typeringsregel wordt automatisch gecontroleerd

```
(<$>) :: (a -> b) -> Parser s a -> Parser s b
```

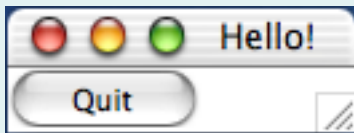
wxHaskell is een portable GUI library voor Haskell.

- ▶ *Portable*: Windows, MacOS X, Linux, FreeBSD, etc.
- ▶ *Simple*: elegante abstracties in Haskell zorgen voor precieze programma's, zeker in vergelijking met equivalente C++ programma's.
- ▶ *Functionaliteit*: gebaseerd op wxWidgets, een serieuze open source C++ library. Gebruikt door AOL communicator en AVG anti-virus.
- ▶ *Onderhoud*: veel mensen werken aan de wxWidgets library, en wxHaskell profiteert daarvan.



wxHaskell demo

```
gui :: IO ()
gui = do f <- frame    [text := "Hello!"]
        q <- button f [text := "Quit"
                        ,on command := close f]
        return ()
```



Internet pagina's

- ▶ www.cs.uu.nl/helium
- ▶ www.haskell.org
- ▶ wxhaskell.sourceforge.net

Referenties

- [1] Bastiaan Heeren and Daan Leijen. Gebruiksvriendelijke compiler voor het onderwijs. *Informatie*, 46(8):46 – 50, Oktober 2004.
- [2] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.