Stichting NIOC en de NIOC kennisbank

Stichting NIOC (www.nioc.nl) stelt zich conform zijn statuten tot doel: het realiseren van congressen over informatica onderwijs en voorts al hetgeen met een en ander rechtstreeks of zijdelings verband houdt of daartoe bevorderlijk kan zijn, alles in de ruimste zin des woords.

De stichting NIOC neemt de archivering van de resultaten van de congressen voor zijn rekening. De website www.nioc.nl ontsluit onder "Eerdere congressen" de gearchiveerde websites van eerdere congressen. De vele afzonderlijke congresbijdragen zijn opgenomen in een kennisbank die via dezelfde website onder "NIOC kennisbank" ontsloten wordt.

Op dit moment bevat de NIOC kennisbank alle bijdragen, incl. die van het laatste congres (NIOC2023, gehouden op donderdag 30 maart 2023 jl. en georganiseerd door NHL Stenden Hogeschool). Bij elkaar bijna 1500 bijdragen!

We roepen je op, na het lezen van het document dat door jou is gedownload, de auteur(s) feedback te geven. Dit kan door je te registreren als gebruiker van de NIOC kennisbank. Na registratie krijg je bericht hoe in te loggen op de NIOC kennisbank.

Het eerstvolgende NIOC vindt plaats op donderdag 27 maart 2025 in Zwolle en wordt dan georganiseerd door Hogeschool Windesheim. Kijk op www.nioc2025.nl voor meer informatie.

Wil je op de hoogte blijven van de ontwikkeling rond Stichting NIOC en de NIOC kennisbank, schrijf je dan in op de nieuwsbrief via

www.nioc.nl/nioc-kennisbank/aanmelden nieuwsbrief

Reacties over de NIOC kennisbank en de inhoud daarvan kun je richten aan de beheerder:

R. Smedinga kennisbank@nioc.nl.

Vermeld bij reacties jouw naam en telefoonnummer voor nader contact.

Functioneel Programmeren met Helium

Bastiaan Heeren en Daan Leijen
Instituut voor Informatica en Informatiekunde, Universiteit Utrecht

Moderne functionele talen zijn mathematisch precies, notationeel elegant, en sterk getypeerd. Deze talen zijn daarom bij uitstek geschikt voor het onderwijzen van programmeerconcepten en algoritmiek. Helium is een onderzoeksproject waarbij een krachtig typesysteem gecombineerd wordt met precieze en gebruiksvriendelijke foutmeldingen.

Keywords: *puur functioneel programmeren, gebruiksvriendelijke foutmeldingen.*

1 Introductie

Pure functionele programmeertalen zijn uitermate geschikt voor het ontwikkelen van correcte software. De eigenschappen die dit mogelijk maken zijn niet alleen van belang bij software ontwikkeling, maar maken deze talen juist ook interessant voor onderwijs doeleinden.

Imperatieve programmeertalen, zoals bijvoorbeeld Java, C, en C^{++}, beschrijven programma's aan de hand van een serie opdrachten. Deze talen genieten een grote populariteit en worden veelvuldig gebruikt en onderwezen. Pure functionele programmeertalen vormen een alternatief paradigma, waarin programma's worden geschreven als een verzameling van functies zoals bekend uit de wiskunde. Dit vereist een andere manier van programmeren, zonder aandacht te schenken aan bijvoorbeeld berekeningsvolgorde of geheugenallocatie. Recursie, polymorfie, en hogere-orde functies spelen binnen deze talen een centrale rol. Daarnaast lenen zij zich uitstekend om eigen datastructuren te definiëren, en om eigenschappen over programma's te bewijzen (bijvoorbeeld met inductie).

Haskell is zo'n functionele programmeertaal (andere voorbeelden zijn ML en Clean). Haskell is een sterk getypeerde taal: typeringsfouten worden al tijdens compilatie gemeld aan de gebruiker. Deze foutmeldingen zijn echter dikwijls van een cryptische aard voor beginnende programmeurs.

Onze ervaringen met het onderwijzen van Haskell waren destijds de aanleiding voor het ontwikkelen van een betere compiler waarmee studenten sneller en met meer plezier functioneel programmeren kunnen leren. Het resultaat is de Helium compiler, en deze heeft de volgende kenmerken:

- Helium bevat een nieuw algoritme voor typeinferentie. Dit algoritme gebruikt een globale constraint-analyse op een typeringsgraaf om fouten preciezer te rapporteren.

- Helium bevat een groot aantal heuristieken om mogelijke verbeteringen aan te geven bij een fout. Voorbeelden hiervan zijn het herordenen van argumenten en het corrigeren van verkeerd gespelde namen. Verder genereert Helium vele waarschuwingen en hints die helpen om potentiële fouten te voorkomen.

- De compiler kan fouten registreren op een centrale server. Dit is gebruikt tijdens de introductiecursus functioneel programmeren om letterlijk duizenden programma's vast te leggen. Deze programma's zijn geanalyseerd om vast te

stellen welke fouten veel gemaakt worden, en om de heuristieken van de compiler te verbeteren.

• Helium ondersteunt bijna volledig Haskell, met als uitzondering enkele taalconstructies die de kwaliteit van de foutmeldingen negatief beïnvloeden. De meest opvallende taalconstructie die ontbreekt is overloading (type-klassen). In de afgelopen jaren hebben we onderzoek verricht naar het verbeteren van foutmeldingen met overloading. De volgende release van Helium zal type-klassen wel ondersteunen.

• Programma's worden uitgevoerd door de lazy virtual machine (LVM). Bij een exceptie wordt niet alleen de exceptie zelf getoond, maar ook alle waarden die deze exceptie propageren. Dit helpt aanzienlijk bij het debuggen van programma's. Naast standaard excepties zoals ontbrekende patronen, controleert de LVM ook op integer overflow en vormen van oneindige recursie.

• Een simpele, maar effectieve grafische omgeving is beschikbaar voor de Helium compiler (zie Figuur 1). Hierbij kan men automatisch naar de locatie van een fout springen in een editor. Ook worden de verschillende soorten fouten en waarschuwingen in duidelijke kleuren weergegeven.

In de volgende twee paragrafen besteden we eerst aandacht aan de basis van functionele talen, om vervolgens specifiek de Helium compiler te beschrijven.

2 Puur functioneel programmeren

In een pure functionele taal bestaat ieder programma uit een verzameling van definities en expressies. In het volgende voorbeeld wordt een functie gedefinieerd die het kwadraat berekent van zijn argument.

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

Een interpreter kan nu expressies evalueren die deze definitie gebruiken.

```
> kwadraat (4 + 4)
64
```

Expressies in dergelijke talen gedragen zich als wiskundige formules. De betekenis van een wiskundige expressie is slechts een waarde, en er is geen verborgen extra effect dat afhankelijk is van de wijze waarop men die waarde afleidt. De expressie `kwadraat 4` beschijft precies dezelfde waarde als de expressie `16` of `XVI`, namelijk het getal zestien. Een geldige expressie kan daarom ook in willekeurige volgorde worden gereduceerd.

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

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

Dit betekent dat we met pure functionele talen equationeel kunnen redeneren over eigenschappen van algoritmen met standaard technieken uit de wiskunde. Imperatieve programmeertalen als C en Java maken dit zo goed als onmogelijk, omdat zij arbitraire effecten toestaan, zoals bijvoorbeeld het rekenen met pointers. Het voordeel van een declaratieve taal als Haskell is dus dat men zich kan concentreren op de essentie, en niet wordt afgeleid door arbitraire details.

Een ander voordeel voor het onderwijs is de mogelijkheid om nieuwe algebraïsche datatypes te introduceren. Met deze definities kunnen de beschikbare datatypes, zoals getallen en letters, verder worden uitgebreid. Een binaire boom met waarden in de bladeren kan als volgt worden opgeschreven.

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

Het nieuwe `Boom` data type bevat waarden van het type `a` en heeft twee constructoren: een `Boom` bestaat ofwel uit een `Blad` met een waar-

de `a`, of een `Tak` met twee deelbomen. Een voorbeeld van een kleine boom met drie getallen is:

```
nummers = Tak (Blad 1)
              (Tak (Blad 2) (Blad 3))
```

Functies over bomen worden inductief gedefiniëerd over de structuur van de constructoren, zoals bijvoorbeeld een functie die de som van alle getallen in een boom berekent, of een functie die de diepte van een boom bepaalt.

```
som (Blad n)     = n
som (Tak t1 t2) = som t1 + som t2

diepte (Blad n) = 0
diepte (Tak t1 t2)
  = 1 + max (diepte t1) (diepte t2)
```

Met behulp van equationeel redeneren kan men nu vrij gemakkelijk allerlei eigenschappen van bomen bewijzen. Zo zal de diepte van een boom altijd kleiner zijn dan het aantal elementen in de boom. In talen als C of Java, waar de bomen met pointers en references zijn geïmplementeerd, zijn dit soort eigenschappen zeer moeilijk te bewijzen. De bovenstaande definities in Haskell zijn bovendien zeer precies: het equivalente programma in bijvoorbeeld Java zal veel meer arbitraire details bevatten, zoals klasse-definities, type-annotaties en *return* statements.

3 Sterke typering

Haskell bevat een geavanceerd typesysteem dat automatisch vele fouten in programma's opspoort. Het systeem is zo krachtig, dat elk getypeerd programma nooit een ongeldige bewerking zal uitvoeren! Alle Haskell waarden zijn verdeeld in verzamelingen die we types noemen. Basistypes zijn bijvoorbeeld gehele getallen (`Int`) en letters (`Char`). Met deze basistypes kunnen we complexere types samenstellen, zoals bijvoorbeeld functies van getallen naar getallen (`Int -> Int`), of bomen die letters bevatten (`Tree Char`). Hier zijn de types van eerder gedefinieerde namen.

```
kwadraat :: Int -> Int
nummers   :: Boom Int
som       :: Boom Int -> Int
diepte    :: Boom a   -> Int
```

Vooral het type van `diepte` is interressant: de kleine letter `a` geeft aan dat deze functie *polymorf* is in de elementen van de boom. De diepte van een boom is natuurlijk alleen bepaald door zijn structuur en onafhankelijk van het type van de elementen! Dit betekent ook dat we deze functie slechts eenmaal hoeven te definiëren, en niet voor ieder elementtype apart. Het is verbazingwekkend dat een natuurlijk concept als polymorfie door zo weinig programmeertalen goed wordt ondersteund.

Een ander voorbeeld van een polymorfe operatie is functiecompositie, bekend uit de wiskunde, om twee functies te combineren tot één nieuwe functie. In Haskell gebruiken we hiervoor de operator `(.)`, die als volgt is gedefinieerd.

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

Nu kunnen we bijvoorbeeld `kwadraat . (+3)` opschrijven: deze functie neemt een getal, telt er drie bij op, en neemt daar vervolgens het kwadraat van. We hoeven ons echter niet te beperken tot functies over getallen. Het type van functiecompositie is daarom dan ook zeer polymorf:

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

Functiecompositie neemt een functie van een type `b` naar een type `c`, en een functie van `a` naar `b`, en levert een nieuwe functie op van `a` naar `c`. In talen zonder parametrisch polymorfisme is het type van deze functie niet uit te drukken. In Java bijvoorbeeld moet men onveilige type *casts* gebruiken met het `Object` type om functiecompositie te kunnen beschrijven.

Voor iedere Haskell expressie kan automatisch een type worden afgeleid. Daarom hoeven we nooit het type van een expressie op te geven, maar kan dit worden bepaald door een zoge-

heten type-inferentie algoritme. Elke expressie waaraan geen geldig type kan worden toegekend wordt niet geaccepteerd door de compiler: dit soort expressies hebben nu eenmaal geen zinvolle waarde. Bijvoorbeeld: met de operator `(==)` kan op gelijkheid getest worden, maar men kan geen appels met peren vergelijken.

```
> 1 == 'a'
error: cannot compare Int with Char
```

Wij geloven dat sterke typering essentieel is voor goed informatica onderwijs: het volgen van een stricte type discipline leidt tot heldere en goed gestructureerde programma's. Bovendien vangt sterke typering ook vele logische fouten af: denkfouten leiden bijna altijd tot typeringsfouten!

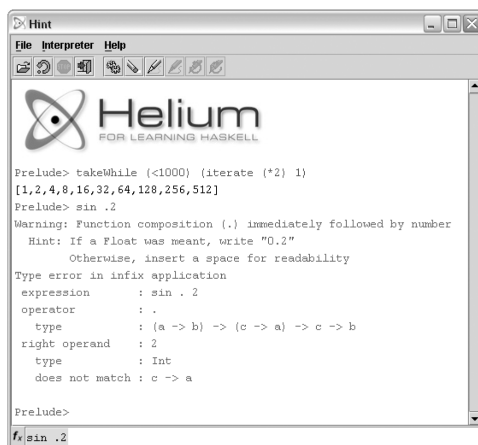
Typesystemen kunnen ook in de weg zitten, zoals elke Java programmeur zal kunnen beamen. Haskell's typesysteem is echter dermate expressief dat dit vrijwel nooit voorkomt. Een belangrijk onderdeel daarvan is de ondersteuning voor polymorfe functies, zoals functiecompositie, en polymorfe datastructuren, zoals `Boom a`. Helaas zijn de foutmeldingen van de bestaande Haskell compilers dikwijls cryptisch van aard en moeilijk te begrijpen. Een voorbeeld van een cryptische melding van de Haskell interpreter Hugs is:

```
> diepte 1
ERROR - Illegal Haskell 98 class
        constraint in inferred type
* Expression : diepte 1
* Type       : Num (Boom a) => Int
```

De Helium compiler is speciaal ontworpen om dit soort cryptische meldingen te voorkomen. In de volgende paragrafen worden verschillende technologische aspecten besproken.

4 De Helium compiler

De twee belangrijkste Haskell implementaties zijn de interpreter Hugs en de Glasgow Haskell Compiler (GHC). De kleinschalige Hugs interpreter is vooral geschikt bij het ontwikkelen van programma's, en wordt veelal gebruikt in het onderwijs. GHC is een "industrial strength" compiler waarmee zeer efficiënte code gegenereerd kan worden, en die tal van uitbreidingen op de taal ondersteunt. In beide gevallen laten de foutmeldingen te wensen over, en hiervoor vormt de Helium compiler een alternatief.



Figuur 1: De Helium interpreter

Het ontwerp van de compiler is afgestemd op het geven van goede foutmeldingen, en verzamelde informatie wordt tijdens het compilatieproces dan ook zorgvuldig bewaard. Helium maakt verder gebruik van een nieuw type-inferentie algoritme dat gebaseerd is op het verzamelen en oplossen van constraints. Deze benadering maakt het mogelijk om niet alleen te constateren dat er een fout gemaakt is, maar ook op zoek te gaan naar de oorzaak hiervan. Maar de compiler onderscheidt zich vooral in de mate waarin er geanticipeerd wordt op veel gemaakte fouten. In de volgende paragrafen kijken we in detail naar de verschillende soorten waarschuwingen en foutmeldingen van Helium.

5 Waarschuwingen en suggesties

De Helium compiler heeft naast echte foutmeldingen ook een groot arsenaal aan waarschuwingen en suggesties. In het algemeen moet men hier zeer voorzichtig mee zijn: niets zeggen kan beter zijn dan het geven van een verkeerde suggestie! Het registratiesysteem kwam hier zeer van pas om vast te stellen wat de meest voorkomende fouten zijn die studenten maken, en de waarschuwingen en suggesties van Helium zijn veelal gebaseerd op deze gegevens. In paragraaf 8 presenteren we het registratiesysteem in meer detail.

Het volgende voorbeeld illustreert het nut van de suggesties en waarschuwingen die door Helium worden gerapporteerd:

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

Het bovenstaande programma wordt door iedere andere Haskell compiler geaccepteerd zonder enige waarschuwing. Helium daarentegen ziet een aantal potentiële problemen:

```
(4,1): Warning: Tab character encountered;
        this may cause problems with layout
  Hint: Configure your editor to
        replace tabs by spaces
(3,1) : Warning: Missing type signature:
         myFIlter :: (a -> Bool) -> [a] -> [a]
(2,10): Warning: Variable "p" is not used
(2,1), (3,1): Warning: Suspicious adjacent
  functions "myFilter" and "myFIlter"
```

We zien dat Helium meerdere waarschuwingen kan geven met precieze locaties in de broncode: regel en kolom. De laatste waarschuwing laat ook zien dat Helium meerdere locaties kan rapporteren die bijdragen aan een melding. In een ontwikkelomgeving kan dit gebruikt worden om tussen de verschillende locaties te navigeren.

De eerste waarschuwing is een typische melding voor een compiler in het onderwijs: fouten in de layout van Haskell programma's vanwege onzichtbare tabs zijn een bron van frustratie voor studenten. De tweede melding moedigt studenten aan om type signaturen op te schrijven bij de functies die ze definiëren. Hiermee stimuleren we een goede gewoonte, en wordt de student gedwongen na te denken over zijn opgeschreven functies. Zo'n waarschuwing hoort juist thuis in een compiler gericht op het onderwijs. In dit specifieke voorbeeld is de waarschuwing overigens veroorzaakt door een schrijffout. De laatste waarschuwing wijst deze moeilijk te vinden fout aan: de naam van de functie in de laatste vergelijking is verkeerd gespeld. Uit ons registratiesysteem blijkt dat dit soort fouten geregeld voorkomen, maar pas na relatief lang *debuggen* gevonden worden.

Helium berekent de *minimale bewerkingsafstand* tussen twee definities die elkaar opvolgen. Een bewerkingsafstand is gespecificeerd als het aantal basisbewerkingen dat men moet uitvoeren om de ene naam in de andere te veranderen. In het bovenstaande geval is de afstand slechts één bewerking: de letter `i` moet in een hoofdletter veranderd worden. Wanneer de afstand klein genoeg is neemt Helium aan dat er een schrijffout is gemaakt en wordt er een waarschuwing gegeven.

Hetzelfde mechanisme wordt gebruikt om schrijffouten in de definitienamen te ontdekken. Hier is een ander programma uit het registratiesysteem:

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

Dit programma bevat twee ongedefinieerde variabelen:

```
(2,10): Undefined variable "maximun"
  Hint: Did you mean "maximum" ?
(2,30): Undefined variable "xs"
```

Helium meldt ons dat `maximun` wel erg lijkt op een andere naam die wel gedefinieerd is, name-

lijk `maximum`.

Een andere veel gemaakte fout van studenten is om floating-point getallen verkeerd op te schrijven.

```
test = sin .2
```

Qua syntax is dit een goed Haskell programma omdat de punt (`.`) gelezen kan worden als functiecompositie. De interpreter Hugs geeft dan ook een verwarrende foutmelding:

```
ERROR "lex1.hs" (line 1):
  Unresolved top-level overloading
* Binding              : test
* Outstanding context : (Floating b,
                         Num (c -> b))
```

Deze foutmelding is het gevolg van de impliciete overloading van integer constanten. Helium geeft natuurlijk ook een typefoutmelding, maar pas na de waarschuwing dat het ongebruikelijk is om een punt voor een getal neer te zetten.

```
(1,13): Warning:
  Function composition (.)
  immediately followed by number
  Hint: If a Float was meant, write "0.2"
        Otherwise, insert a space for
        readability

(1,13): Type error in infix application
 expression     : sin . 2
  operator      : .
  type          : (a -> b) -> (c -> a) -> c -> b
 right operand  : 2
  type          : Int
  does not match: c -> a
```

Natuurlijk zijn zulke suggesties veel waard voor de beginnende Haskell programmeur. Studenten zullen zich sneller de syntax eigen maken, en kunnen dus meer aandacht besteden aan de werkelijke leerdoelen

6 Syntaxfouten

Zoals het voorgaande voorbeeld laat zien, kan men snel (te) veel tijd besteden aan het aanleren van syntax, in plaats van de essentiële concepten van puur functioneel programmeren. Helium probeert daarom uitgebreide syntactische foutmeldingen te geven; dit heeft slechts weinig prioriteit in productie compilers. Een programma wordt eerst lexicaal geanalyseerd door de balancering van haakjes te controleren. Hierbij worden vele fouten afgevangen die anders leiden tot cryptische syntaxfouten.

```
test :: [(Int, String)]
test = [(1, "one"), (2, "two"), (3, "three")
```

In het bovenstaande voorbeeld is de programmeur vergeten de lijst af te sluiten met een sluitteken (`]`). Omdat de balancering van haakjes nauwgezet wordt bijgehouden kan Helium hiervoor een exacte foutmelding produceren:

```
(2,8): Bracket '[' is never closed
```

Hier is een duidelijk verschil zichtbaar met de melding van de GHC compiler. Deze houdt geen kolom informatie bij en is zeer imprecies in de locatie van de fout. Verder wordt er ten onrechte gesuggereerd dat dit te maken heeft met verkeerde indentatie:

```
syn3.hs:3: parse error
  (possibly incorrect indentation)
```

Voor het bovenstaande voorbeeld geeft de interpreter Hugs een interessante melding omdat het naar een niet bestaand sluitteken verwijst. Dit teken is door de compiler *zelf* toegevoegd om de layout van Haskell te analyseren.

```
ERROR "syn3.hs" (line 3):
  Syntax error in expression
  (unexpected `}', possibly due to bad layout)
```

De grammaticale structuur wordt geanalyseerd met behulp van geavanceerde parsertechnologie, genaamd Parsec. Deze technologie houdt niet alleen de positie van syntaxfouten bij, maar ook alle syntax-constructies die op dat punt van de invoer geldig waren geweest.

```
remove :: Int -> [Int] -> [Int]
remove n [] = []
remove n (x:xs)
    | n = x     = rest
    | otherwise = x : rest
 where rest = remove n xs
```

In de vierde regel van het bovenstaande voorbeeld worden `n` en `x` vergeleken met `=`, terwijl hiervoor de operator `==` gebruikt had moeten worden. Helium merkt de fout op zodra de tweede `=` wordt gezien.

```
(4,16): Syntax error:
    unexpected '='
    expecting expression, operator,
    constructor operator, '::', '|',
    keyword 'where', or end of block
    (based on layout)
```

Contrasteer dit met de melding van de GHC compiler:

```
syn4.hs:4: parse error on input `='
```

Helium laat duidelijk de locatie van de fout zien, terwijl het in de foutmelding gegeven door GHC onduidelijk is welke van de twee `=` tekens verantwoordelijk is voor de fout. Verder laat Helium zien welke sytax constructies wel geldig zijn op dat punt van de invoer.

7 Typeringsfouten

Het registratiesysteem laat zien dat veruit de meeste fouten die gemaakt worden typeringsfouten zijn. Zoals gezegd infereert Helium automatisch de types van geldige expressies, en wordt er een melding gegeven indien er een fout gemaakt is. Echter, om extra ondersteuning voor foutmeldingen te kunnen bieden wijkt het type-inferentie mechanisme sterk af van de traditionele benadering. De analyse is geformuleerd als een constraint probleem om zoveel mogelijk informatie vast te houden tijdens het inferentieproces. Aan de hand van een verzameling constraints gegenereerd voor een programma wordt er een graaf opgebouwd die de samenhang tussen expressies en types weergeeft. Inconsistenties in de graaf worden opgelost door het inzetten van een verzameling heuristieken om veel voorkomende patronen te detecteren, en hier geschikte meldingen voor te geven.

Eén heuristiek voor het oplossen van een type-inconsistentie is het tellen van het aantal constraints die een bepaald type prefereren. Als er bijvoorbeeld drie indicaties zijn dat een expressie type `Int` moet hebben, maar slechts één indicatie dat het een `Bool` moet zijn, dan zal de foutmelding zich concentreren op de uitzonderlijke `Bool`.

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

In het bovenstaande programma bevatten de `if` takken een `Bool` en een `Int` expressie. Vanwege de typesignatuur heeft Helium meer aanwijzingen dat dit `Int` moet zijn, en de foutmelding geeft weer dat de `True` expressie fout is:

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

Traditionele type-inferentie neemt dergelijke constraints niet mee en zal bijvoorbeeld een foutmelding geven over de andere tak. Het globaal oplossen van de verzameling constraints voorkomt een van-links-naar-rechts afwijking die typisch voorkomt in standaard unificatie-algoritmen.

Met behulp van de typeringsgraaf kan Helium een aantal strategieën toepassen om inconsistenties op te lossen. Als dit lukt kan Helium een suggestie geven om het inzicht in de gemaakte fout te vergroten. Om misleidende suggesties te voorkomen geven we alleen een suggestie als deze een unieke oplossing biedt. Voorbeelden van strategieën zijn het verwisselen van functie argumenten die in de verkeerde volgorde gegeven zijn, of het invoegen van een vergeten argument. Neem bijvoorbeeld het volgende programma:

```
test = map [1..10] even
```

De hogere-orde functie `map` verwacht twee argumenten: een functie en een lijst. De student heeft de argumenten van `map` echter in de ver-

keerde volgorde gezet. Helium anticipeert op deze veelvuldig gemaakte fout.

```
(1,8): Type error in application
 expression      : map [1 .. 10] even
 term            : map
  type           : (a -> b)-> [a]          -> [b]
  does not match: [Int]    -> (Int -> Bool)-> c
 probable fix    : re-order arguments
```

Helium gebruikt een variant van de minimale bewerkingsafstand om te bepalen hoe expressies veranderd kunnen worden om de type-inconsistenties in de typeringsgraaf op te lossen. Als dit een unieke oplossing biedt bij weinig bewerkingen, zoals het herordenen van functie argumenten, dan geeft Helium een suggestie. Merk ook op dat het typesignatuur van `map` is gegeven in de foutmelding, en dat dit mooi is uitgelijnd met het geïnfereerde type. GHC verwijst slechts naar een willekeurig argument van `map`:

```
tp4.hs:1:
 Couldn't match 'a -> b' against '[t]'
    Expected type: a -> b
    Inferred type: [t]
 In an arithmetic sequence: [1 .. 10]
 In the first argument of 'map',
    namely '[1 .. 10]'
```

Een andere belangrijke strategie beschouwd *siblings*: dit zijn semantisch gerelateerde functies met enigszins afwijkende types. Zo zijn er bijvoorbeeld twee functies beschikbaar om het maximum te bepalen:

```
max     :: Int -> Int -> Int
maximum :: [Int] -> Int
```

De ene werkt op twee getallen, de andere op een lijst met getallen. Omdat deze functies zo verwant zijn aan elkaar maken we er siblings van. Op een soortgelijke manier relateren we ook integer constanten aan floating-point getallen. Helium ondersteunt speciale directieven waarmee dit soort sibling paren kunnen worden opgeschreven:

```
sibling   max , maximum
```

Het volgende voorbeeld laat de sibling strategie in actie zien. Het programma is geschreven door een student als onderdeel van het werkcollege. Meerdere studenten maakten daar dezelfde fout.

```
maxLength :: [String] -> Int
maxLength xs = max (map length xs)
```

Deze definitie is bijna correct, alleen heeft de student de verkeerde functie gebruikt om het maximum te bepalen. De Hugs interpreter geeft de volgende melding:

```
ERROR "A.hs":2 -
    Type error in explicitly typed binding
*** Term            : maxLength
*** Type            : [String] -> [Int] -> [Int]
*** Does not match : [String] -> Int
```

Omdat het type van `max` normaal gesproken overloaded is kan Hugs deze definitie typeren, al komt het geïnfereerde type niet overeen met de opgeschreven typesignatuur. De GHC compiler is al iets beter in zijn foutmelding:

```
A.hs:2:
    Couldn't match 'Int' against 'a -> a'
        Expected type: Int
        Inferred type: a -> a
    Probable cause:
        'max' is applied to too few arguments
        in the call (max (map length xs))
    In the definition of 'maxLength':
        max (map length xs)
```

Omdat Helium een typeringsgraaf gebruikt, kan deze opmerken dat de sibling functie van `max` de inconsistentie uniek kan oplossen. Helium geeft daarom de suggestie om `maximum` te gebruiken:

```
(2,16): Type error in variable
 expression       : max
   type           : Int    -> Int -> Int
   expected type  : [Int] -> Int
 probable fix     : use maximum instead
```

Deze melding is niet alleen makkelijk te begrijpen vanwege de sibling, maar ook omdat het type van `max` vermeld wordt.

8 Het registratiesysteem

Tijdens de introductiecursus functioneel programmeren aan de Universiteit Utrecht hebben we een registratiesysteem ingezet om studenten-

programma's op te slaan. Het doel hiervan is om inzicht te krijgen in het typische programmeergedrag van beginnende programmeurs, en om met het verkregen inzicht de compiler nog beter aan te laten sluiten bij de behoeften van deze doelgroep. Gedurende de looptijd van het vak is ieder programma dat gecompileerd wordt vanaf de studentenmachines opgeslagen in een database. Expressies die tussentijds in de Hint interpreter worden geëvalueerd zijn buiten beschouwing gelaten. Dit heeft in de afgelopen twee jaar geleid tot een database van ruim 60.000 programma's, zowel correcte als incorrecte. Om dit te bereiken hebben we de Helium compiler voorzien van de mogelijkheid om contact te zoeken met een centrale server. Deze server legt de opgestuurde programma's vast.

Om de samenhang tussen de programma's in deze database te behouden wordt bij ieder programma historische informatie opgeslagen. Dit bestaat uit de naam van de student, het tijdstip van compilatie, en het versienummer van de compiler. Dit maakt het mogelijk om de voortgang van een individu te traceren en analyseren. Vanwege de precisie in de registratie hebben we enkele maatregelen genomen om de privacy van de studenten te kunnen garanderen. Ten eerste zijn alle studenten vooraf ingelicht over het experiment, en bestond er de mogelijk om niet geregistreerd te worden. Ook is er afgesproken dat de geregistreerde gegevens in geen enkel opzicht van invloed zouden zijn bij de beoordeling van het praktikum. Tenslotte is de database na afloop van het vak geanonimiseerd.

De opgeslagen gegevens vragen om een uitgebreide analyse, en hier wordt momenteel aan gewerkt. Om toch een idee te geven over de mogelijkheden van zo'n analyse presenteren we enkele gegevens uit de dataset van het cursusjaar 2003/2004. De volgende tabel geeft per week het aantal registraties weer.

wk.	6	7	8	9	10	11	12	13
reg.	3511	3649	3263	901	6279	1590	1962	2157

De negende week was lesvrij, en de praktikum deadlines lagen aan het einde van week 10 en week 13 (zoals wellicht ook is af te leiden uit de data). Alle registraties kunnen we indelen in compilatie categorieën: deze komen overeen met de verschillende fasen in een compiler die een programma moet doorlopen. We geven percentages per categorie voor week 6, en voor het gehele praktikum.

compilatie categorie	*week 6*	*totaal*
lexicale fout	3%	3%
syntaxfout	14%	9%
statische fout	8%	10%
typeringsfout	30%	32%
compileerbaar	45%	46%

Gelukkig blijkt bijna de helft van de programma's "correct" te zijn. Hieronder vallen echter ook programma's die een denkfout bevatten en dus semantisch gezien onjuist zijn, en programma's waarin Helium potentiële fouten heeft ontdekt. De significante verschuiving van het percentage syntactisch incorrecte programma's bevestigt het vermoeden dat de eerste hindernis om een programmeertaal te leren het bekend raken met de syntax is. Tenslotte geven we een overzicht van de verdeling van het soort statische fouten. Een programma kan meerdere van dit soort fouten bevatten.

statische fout	*aantal*	*(%)*
ongedefinieerde variabele	2240	55,46%
ongedefinieerde constructor	377	9,33%
typesignatuur zonder functie	368	9,11%
ariteit bij constructor	223	5,52%
ariteit bij functiedefinitie	209	5,17%
ongedefinieerde typeconstr.	196	4,85%
dubbele definitie	157	3,89%
overige fouten	269	6,66%

Opvallend genoeg is het refereren aan een onbekende variabele met afstand de meest gemaak-

te vergissing. Dit onderstreept nogmaals de importantie van het berekenen van een minimale bewerkingsafstand voor ongedefinieerde variabelen.

9 Conclusie

We hebben laten zien dat functionele talen (zoals bijvoorbeeld Haskell) zeer geschikt zijn om de essentie van programmeren te onderwijzen. Dat declaratieve talen en onderwijs goed samengaan wordt tevens aangetoond door het Pan# project. In deze leeromgeving kunnen functionele afbeeldingen en animaties beschreven worden door middel van declaratieve definities en functionele abstracties. Dit systeem wordt gebruikt op middelbare scholen in Amerika om de basis van algebra uit te leggen op een simpele en interactieve manier. Een tweede voorbeeld is de DrScheme programmeeromgeving, waarmee studenten stapsgewijs kennis kunnen maken met de functionele taal Scheme.

Helium kan een waardevolle ondersteuning bieden bij het leren van de programmeertaal Haskell. De gerapporteerde foutmeldingen zijn duidelijk en precies, en door het geven van waarschuwingen wordt een goede programmeerstijl bevorderd. Bovendien wordt er geanticipeerd op een aantal karakteristieke fouten. De compiler is vrij beschikbaar op de Helium website:

http://www.cs.uu.nl/helium

Helium richt zich uitsluitend op het leren van de programmeertaal Haskell. Voor meer geavanceerde toepassingen zal men de overstap naar een compiler moeten maken met meer functionaliteit en bibliotheken, zoals bijvoorbeeld GHC. Hier heeft men ook de beschikking over de wxHaskell library om grafische applicaties te maken. Deze groots opgezette bibliotheek voor grafische user interfaces is gebaseerd op wxWidgets, een uiterst succesvol cross-platform en open-source GUI framework. Na het leren van de essentiële principes van programmeren met Helium, kan men zo aan de slag om grote interactieve applicaties te ontwikkelen.

Onze dank gaat uit naar Arjan van IJzendoorn, die tweeënhalf jaar geleden is begonnen met het ontwerpen van de compiler. Jurriaan Hage heeft een belangrijke rol gespeeld in het opzetten van het registratiesysteem, en het uitvoeren van de beschreven experimenten. Tenslotte bedanken we Rijk-Jan van Haaften en Arie Middelkoop voor hun geleverde bijdrage aan het project.

Referenties

Findler, Robert Bruce *et al.* (2002). DrScheme: A programming environment for Scheme. *Journal of functional programming*, **12**(2), 159–182.

Heeren, Bastiaan, & Leijen, Daan. (2004). Gebruikersvriendelijke compiler voor het onderwijs. *Informatie*, **48**(6).

Heeren, Bastiaan, Leijen, Daan, & van IJzendoorn, Arjan. (2003a). Helium, for learning Haskell. *Pages 62 – 71 of: ACM sigplan 2003 Haskell workshop*. New York: ACM Press.

Heeren, Bastiaan, Hage, Juriaan, & Swierstra, S. Doaitse. 2003b (Aug.). Scripting the type inference process. *International conference on functional programming (ICFP'03)*.

Leijen, Daan. (2004). wxHaskell. *ACM sigplan 2004 Haskell workshop*. Snowbird, Utah, USA: ACM Press.

Peterson, John. 2002 (October). A language for mathematical visualization. *Proceedings of FPDE'02: Functional and declarative languages in education*.

Peyton Jones, Simon, & Hughes, John (eds.). 1998 (Feb.). *Report on the language Haskell'98*.