Gebruiksvriendelijke compiler voor het onderwijs

Helium moet functioneel programmeren ondersteunen

Foutmeldingen van compilers zijn vaak moeilijk te interpreteren. Om het programmeeronderwijs te verbeteren wordt aan de Universiteit Utrecht een compiler ontwikkeld die precieze, gebruikersvriendelijke foutmeldingen genereert.

Bastiaan Heeren en Daan Leijen

De syntax van de programmeertaal Haskell is elegant. De taal is gebaseerd op goede wiskundige principes en alle programma's zijn sterk getypeerd. Dit houdt in dat een expressie een vaststaand type heeft, bijvoorbeeld een lijst van karakters of een getal. Hiermee staat ook vast welke bewerkingen op de expressie mogelijk zijn. Deze eigenschappen maken de taal bijzonder geschikt om programmeerconcepten en algoritmiek te onderwijzen.
Bij talen met een geavanceerd typesysteem zijn compilerfoutmeldingen dikwijls moeilijk te interpreteren. Helium is een compiler voor de taal Haskell, die de auteurs aan de Universiteit Utrecht ontwikkelen. Om programmeeronderwijs te vergemakkelijken, wordt in deze compiler een krachtig typesysteem gecombineerd met precieze en gebruikersvriendelijke foutmeldingen.

Functioneel

Een programma in een functionele taal bestaat uit een verzameling van definities en expressies. Deze worden door de programmeur samengevoegd om een bepaald probleem op te lossen. De computer reduceert deze expressies en is zo een veredelde calculator, met dit verschil dat nieuwe functies en datatypen toegevoegd kunnen worden. Een functiedefinitie die het kwadraat berekent van zijn argument ziet er bij voorbeeld als volgt uit:

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

Na de definitie van een functie kan een *interpreter* expressies evalueren die deze functie gebruiken.

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

De interpreter behandelt de expressie als een wiskundige formule. Een wiskundige expressie stelt een waarde voor; er is geen verborgen effect dat afhankelijk is van de wijze waarop deze waarde wordt afgeleid. De expressie (het kwadraat van 4) beschrijft dezelfde waarde als de expressie '16' of 'XVI', namelijk het getal zestien. Een geldige expressie kan 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
```

Samenvatting

De auteurs doen onderzoek naar compilers die precieze en gebruikersvriendelijke foutmeldingen genereren, voor ondersteuning van onderwijs in programmeren. Samen met anderen ontwikkelden zij Helium, een compiler voor Haskell. Helium gaat uit van een aantal goede gewoontes en levert suggesties bij een aantal veel voorkomende fouten.

Omdat met pure functionele talen standaard technieken uit de wiskunde gebruikt kunnen worden, is het mogelijk hierin equationeel te redeneren over eigenschappen van algoritmen. Met *imperatieve* programmeertalen als C en Java is dit zo goed als onmogelijk, omdat zij arbitraire effecten toestaan, zoals het rekenen met pointers. Met een declaratieve taal als Haskell kan men zich concentreren op de essentie, zonder te worden afgeleid door arbitraire details van een imperatieve programmeerstijl.

Nieuwe typen

Een ander voordeel van Haskell is de mogelijkheid om nieuwe algebraïsche datatypen te introduceren. Deze definities breiden de beschikbare verzameling aan datatypen zoals getallen en letters uit. Een binaire boomstructuur met waarden in de bladeren kan als volgt worden opgeschreven.

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

Het nieuwe boomdatatype bevat waarden van het type a en heeft twee constructoren: een boom bestaat ofwel uit een *Blad* met een waarde van het type a, of uit een *Tak* met twee deelbomen. Hieronder volgt een voorbeeld van een boom met drie getallen (met het type *Boom Int*).

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

Functies over bomen worden inductief gedefinieerd over de structuur van de constructoren, zoals bijvoorbeeld de 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)
```

De bovenstaande definities vormen ook een specificatie: voor alle alternatieven (*Tak* en *Blad*) bepalen we de bijbehorende waarde, waarbij de volgorde van het berekenen niet van belang is. Dit is de essentie van *declaratief* programmeren. Met behulp van equationeel redeneren zijn nu vrij gemakkelijk allerlei eigenschappen van bomen te 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* worden geïmplementeerd, zijn zulke eigenschappen moeilijk te bewijzen. Definities in Haskell zijn bovendien zeer precies: het equivalente programma in Java zal meer arbitraire details bevatten, zoals klassedefinities, type-annotaties en *return statements*.

Sterke typering

Haskell bevat een zeer geavanceerd typesysteem dat automatisch veel fouten in programma's opspoort. Een getypeerd Haskell-programma zal nooit een ongeldige bewerking uitvoeren. Alle Haskell-waarden zijn verdeeld in verzamelingen, die we typen noemen. Basistypen zijn bijvoorbeeld gehele getallen (*Int*) en letters (*Char*). Met deze basistypen kunnen we complexere typen samenstellen, zoals bijvoorbeeld functies van getallen naar getallen (*Int -> Int*), of bomen die letters bevatten (*Boom Char*). De typen van de eerder gepresenteerde definities zien er als volgt uit.

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

Het type van 'som' laat zien dat deze functie alleen kan worden toegepast op bomen met getallen (*Boom Int*). Bij 'diepte' is dat anders. De diepte van een boom wordt alleen bepaald door zijn structuur en is onafhankelijk van het type van de elementen. We zeggen dat 'diepte' *polymorf* is in de elementen van de boom. Dit kom tot uit-

drukking in het type van ‘diepte’, dat geldt voor bomen van elke elementtype, namelijk *Boom a*. Polymorfie is een natuurlijk concept, en bevordert bovendien het hergebruik van code: één definitie kan nu worden hergebruikt voor verschillende typen.
Voor iedere expressie kan automatisch een type worden afgeleid. Dit maakt het overbodig het type van een expressie op te geven. Dit kan namelijk worden bepaald door een zogeheten type-inferentie-algoritme. Een expressie waaraan geen geldig type kan worden toegekend, wordt niet geaccepteerd door de compiler: zo’n expressie heeft geen zinvolle waarde en is dus illegaal. Hierdoor is het uitgesloten dat appels met peren worden vergeleken.

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

Een consequentie van sterke typering is dat een geldig programma nooit een ‘illegale’ bewerking kan uitvoeren zoals het geheugen overschrijven of methoden van *null*-objecten aanroepen. Dit heeft als consequentie dat een correct getypeerd programma nooit een ongeldige bewerking zal uitvoeren. Een dergelijk programma kan wel tot incorrecte resultaten leiden, maar die zijn altijd het gevolg van denkfouten, en niet van ongeziene neveneffecten en andere valkuilen in de taal.

Structuur

Sterke typering is essentieel voor goed informaticaonderwijs: het volgen van een strikte typedisciplineleidt 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 kan beamen. Haskell’s typesysteem is echter dermate krachtig dat dit vrijwel nooit voorkomt. Een belangrijk onderdeel daarvan is de ondersteuning voor polymorfe functies zoals *diepte*, en polymorfe datastructuren zoals *Boom a*. Helaas zijn de foutmeldingen van Haskell-compilers dikwijls cryptisch van aard en moeilijk te begrijpen. Zo’n cryptische melding van de Haskell-interpreter Hugs ziet er bijvoorbeeld uit als hieronder.

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

De Helium-compiler die wordt ontwikkeld aan de Universiteit Utrecht is speciaal ontworpen om Haskell mee te leren. Het type-inferentie-algoritme is zo ontwikkeld dat het veel beter in staat is om goede foutmeldingen te geven. Dit maakt Helium geschikter voor gebruik in het onderwijs.

De compiler

De Helium-compiler is in relatief korte tijd ontwikkeld aan de Universiteit Utrecht en is geschreven in Haskell. Naast de *command-line compiler* is er een simpele maar effectieve interpreter (in Java), Hint genaamd. Deze gebruikt verschillende kleuren om de soorten fouten en

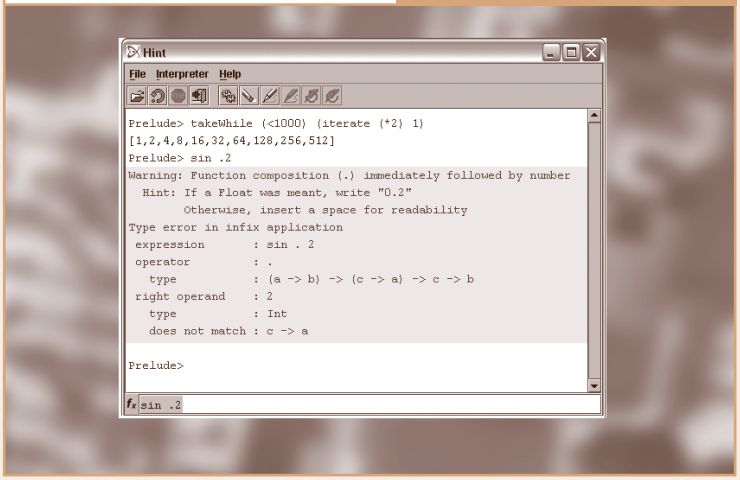
waarschuwingen te onderscheiden. Verder biedt de interpreter ondersteuning om direct naar een fout te springen in een tekst-*editor* naar keuze, wat het programmeergemak aanzienlijk vergroot. Helium is nu drie maal ingezet binnen de introductiecursus Functioneel Programmeren aan de Universiteit Utrecht. Om meer inzicht te verkrijgen in het programmeergedrag van beginners is er een registratiesysteem opgezet dat tijdens de cursussen programma's van studenten opslaat. Dit heeft geleid tot een database van ruim 50.000 programma's. Met deze database kunnen we de foutmeldingen nog verder aanpassen aan de behoeften van de gebruikers. Naast de verzameling programma's is er ook historische informatie opgeslagen die relevant kan zijn om het programmeerproces van een individu te bestuderen.

Foutmeldingen

Het geven van duidelijke foutmeldingen heeft een centrale rol gespeeld in zowel het ontwerp als de implementatie van de compiler. Helium ondersteunt (standaard) Haskell bijna volledig, met als uitzondering enkele taalconstructies die de kwaliteit van de foutmeldingen negatief beïnvloeden. Het opvallendste verschil is het ontbreken van typeklassen. Hoewel typeklassen nuttig zijn, zijn ze niet fundamenteel voor functionele talen, en het weglaten ervan verbetert de foutmeldingen significant.
De compiler kan meerdere foutmeldingen tegelijkertijd rapporteren en beschikt over een wijd scala aan waarschuwingen en hints. Deze kunnen de aandacht vestigen op potentiële gevaren. Zo wordt bijvoorbeeld bij een functie als *diepte* gecontroleerd of deze is gedefinieerd voor alle mogelijke gevallen (zowel *Blad* als *Tak*). Aan de andere kant helpen de waarschuwingen een goede programmeerstijl te ontwikkelen. Zo wordt het declareren van het type van een functie beschouwd als een goede gewoonte. Dit stimuleren we door een melding te genereren als dit niet gebeurt, ook al kan het type automatisch worden afgeleid.

Als het mogelijk is, worden meldingen voorzien van suggesties om het programma te verbeteren. Deze adviezen zijn gebaseerd op observaties van problemen die studenten ondervinden tijdens practica. In de expressie (*sin .2*) wordt de punt bijvoorbeeld geïnterpreteerd als functiecompositie. Deze expressie wordt terecht afgewezen omdat de constante '2' geen functie is. Helium zal suggereren om '0.2' op te schrijven voordat de typeringsfout wordt getoond (zie figuur 1). Een andere veelgemaakte fout is het verkeerd refereren aan een definitie als de programmeur een tikfout maakt. Helium gebruikt een '*shortest edit distance*'-algoritme om namen van beschikbare definities op te sommen die licht afwijken van de genoemde naam, om zo de beoogde naam te vinden en deze aan de programmeur aan te bieden.

Suggestie voor verbetering 1



Hints

Er is veel aandacht besteed om typeringsfoutmeldingen inzichtelijk te maken. Bij een typefout berekent Helium de minimale verandering die nodig is om een programma typecorrect te krijgen, en geeft dit als hint terug aan de gebruiker. Een hint kan zijn om de integer constante *7* te vervangen door de overeenkomstige *floating point* constante. Een veelvoorkomende fout is het verwisselen van functieargumenten. De compiler zal de suggestie geven om argumenten te verwisselen als de typefout daarmee is opgelost. Ook kan de programmeur het advies krijgen om een functie te vervangen door een semantisch gerelateerde functie (met een verschillend type) die wel past in de context. Zo bevat Haskell's standaardbibliotheek een operator om een element aan een lijst toe te voegen, en een operator om twee lijsten te combineren. Deze worden met grote regelmaat door studenten verwisseld. Met de suggesties van Helium kunnen studenten makkelijker hun fouten corrigeren.

Om extra ondersteuning voor foutmeldingen te kunnen bieden, wijkt het type-inferentie-algoritme in Helium af van de traditionele benadering. Het algoritme wordt geformuleerd als een *con-*

straintprobleem, om tijdens het de uitvoering van het algoritme zoveel mogelijk informatie vast te houden. Aan de hand van de gegenereerde verzameling constraints bouwt Helium voor ieder programma een graaf op die de samenhang tussen expressies en typen 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. Ten slotte kan men ook met speciale directieven de typeringsfoutmeldingen van buitenaf beïnvloeden en inrichten naar eigen smaak.

Literatuur

Bird, R., & P. Wadler (1998). *Introduction to Functional Programming using Haskell (second edition)*. Prentice Hall PTR.

Heeren, B., D. Leijen & A. van IJzendoorn (2003). Helium, for Learning Haskell. In: *Proceedings of the ACM Haskell Workshop*.

Heeren, B., J. Hage, & S.D. Swierstra (2003). Scripting the Type Inference Process. In: *Proceedings of the ACM International Conference on Functional Programming*.

Hudak, P. (2000). *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press.

Leijen, D. (2004). wxHaskell: A portable and concise GUI library for Haskell. In: *Proceedings of the ACM Haskell Workshop*.

Thompson, S. (1999). *Haskell: The Craft of Functional Programming (second edition)*. Addison-Wesley.

Links

Haskell: www.haskell.org
Helium: www.cs.uu.nl/helium
Pan#: www.haskell.org/edsl/pansharp.html
wxHaskell: wxhaskell.sourceforge.net

Bastiaan Heeren
is assistent in opleiding aan het Institute of Information and Computing Sciences van de Universiteit Utrecht. E-mail: bastiaan@cs.uu.nl.

Daan Leijen
is postdoctoraal onderzoeker aan het Institute of Information and Computing Sciences van de Universiteit Utrecht. E-mail: daan@cs.uu.nl.

Opslag

IBM stap verder in nanotechnologie

Wetenschappers van IBM hebben een fundamentele eigenschap van een atoom gemeten: de energie die nodig is om de magnetische oriëntatie te wijzigen. De onderzoekers in het 'spintronics'-project gaan kijken wat er gedaan kan worden met de neiging van elektronen rond te tollen ('spin'), een eigenschap die al in de jaren twintig is ontdekt. Elektronen draaien om een atoomkern heen, maar ook om hun eigen as. De richting waarin dat gebeurt, bepaalt hun magnetische oriëntatie, te weten 'omhoog' of 'omlaag'.
Door die 'spin' te manipuleren kan op nanoschaal informatie worden opgeslagen. Spintronics-technologie is al verwerkt in de gmr-schijfleeskoppen waarmee IBM sinds 1997 de capaciteit van harde schijven flink heeft verhoogd.
Maar IBM wil een stuk verder gaan; de onderzoekers zien sensoren, supergeleiders en zelfs processors op basis van spintronics aan de horizon, een ontwikkeling die 'een revolutie in elektronica' zou betekenen. Elektronica zou dan immers worden gebaseerd op de manier waarop elektronen rondtollen, in plaats van hoe ze elektrische stromen doorgeven. Enen en nullen worden niet langer bepaald door de beïnvloeding van stroomsterkte en spanning, maar door de 'spin' van elektronen te manipuleren met magnetische velden.

www.ibm.com

Elektronen van een elektronenmicroscoop (boven) bestralen een atoom (cirkel in het midden). Als de energie een drempelwaarde overschrijdt wijzigt de magnetische orientatie, bijvoorbeeld van omhoog naar omlaag (wat wordt weergegeven door de pijl in de cirkel).