Koen Pauwels

Projectvoorstel

Project: Implementatie van een zuiver functionele programmeertaal

Begeleider: Prof. dr. Dirk Janssens

Motivatie

Ik ben al geruime tijd geïnteresseerd in zuiver functioneel programmeren, en meer specifiek in Haskell.

Functionele talen zijn talen die het belang van zuivere functies sterk benadrukken. Zuivere functies zijn functies die geen neveneffecten hebben en waarvan de teruggegeven waarde steeds louter afhankelijk is van de argumenten.

De motivatie hiervoor is dat het makkelijk is om met zuivere functies te redeneren. Hoewel het initieel ontwerp van de code soms moeilijker is wanneer we mutatie moeten vermijden, lijkt zuiver functionele code resistenter tegen bepaalde soorten bugs en lijkt ze makkelijker te onderhouden (zeker in combinatie met sterke typering).

Haskell voegt daar ook nog *lazy evaluation* aan toe: een expressie wordt pas geëvalueerd wanneer dat strikt noodzakelijk is. Deze evaluatiestrategie is niet inherent aan functioneel programmeren, maar wel enkel van toepassing op zuiver functionele code. Het is namelijk vrijwel onmogelijk om uitgestelde evaluaties en neveneffecten met elkaar te combineren zonder dat het programma onbegrijpelijk wordt.

Een belangrijk nadeel is dat de performantie van imperatieve talen moeilijk te evenaren is. Functionele programma's maken meestal minder gebruik van arrays omdat deze te duur zijn om niet-destructief te updaten. De datastructuren die dan ter vervanging gebruikt worden (vaak boomstructuren of gelinkte lijsten) hebben een zekere extra overhead en een slechtere ruimtelijke lokaliteit. Lazy evaluation maakt het daarbovenop ook moeilijker om voorspellingen te doen over het ruimteverbruik van de code.

Deze problemen kunnen deels worden opgevangen doordat de code makkelijker te paralleliseren is, en door optimalisatie door de compiler. Omdat de compiler bepaalde garanties krijgt over zuivere code, kan deze soms agressiever optimaliseren dan een compiler voor een imperatieve taal.

Om performante Haskell code te schrijven is het dus belangrijk om te begrijpen welke optimalisaties een compiler kan uitvoeren.

Ik zou graag het individueel project gebruiken om meer bij te leren over hoe een zuiver functionele taal geïmplementeerd wordt. Verder zou ik graag meer ervaring krijgen in het schrijven van een niet-triviaal programma in zuiver functionele stijl.

Mijn voorstel is dus om een partiële Haskell implementatie te schrijven in Haskell.

Concreet

Dit voorstel betreft een zelfstudie van het boek “The Implementation of Functional Programming Languages” van Simon Peyton Jones (1987).

Als eindresultaat zal minimaal een werkende, partiële implementatie van de Haskell programmeertaal worden opgeleverd. Verder zal er documentatie worden opgeleverd die moet helpen om de evaluatie vlot te doen verlopen, en verslagen die de theorie van het boek linken aan de implementatie.

Met partieel wordt bedoeld: de implementatie zal enkel zuivere code kunnen evalueren, er zal geen input/output systeem worden voorzien. De nadruk van de uitbreidingen bovenop de minimale implementatie zal voornamelijk liggen op zuivere code meer performant evalueren.

Verder zullen er ongetwijfeld nog veel vereisten van de Haskell standaard niet worden geïmplementeerd, bijvoorbeeld tail call elimination en garbage collection worden niet in de basisvereisten opgenomen.

Een programma voor deze implementatie zal dus bestaan uit een lijst van functie- en typedefinities, met 1 enkele expressie die geëvalueerd wordt, waarna het resultaat wordt uitgeprint.

Voorgestelde basisvereisten

De vereisten die hier worden opgelijst zouden voldoende moeten zijn om te slagen voor het vak.

- Omzetting van Haskell naar verrijkte lambdacalculus (hoofdstukken 2 en 3).
- Algebraïsche datastructuren en pattern matching (hoofdstukken 4 en 5).
- Verrijkte lambdacalculus omzetten naar lambdacalculus (hoofdstuk 6).
- Type checking en type inference (hoofdstukken 8 en 9).
- Graafvoorstelling en graafreductie (hoofdstukken 10, 11 en 12).

Uitbreidingen

Het is vrijwel zeker onmogelijk om alle uitbreidingen uit het hele boek te implementeren. Sommige hoofdstukken, zoals bijvoorbeeld dat over garbage collection, bevatten slechts een oppervlakkige introductie tot het onderwerp, en vallen waarschijnlijk buiten de scope van dit project.

Hier volgt een volledige lijst van mogelijke uitbreidingen, met in het vetgedrukt de uitbreidingen waarvan ik vermoed dat het realistisch is om ze op te leveren.

- **List comprehensions (hoofdstuk 7).**

- Compilatie
  - **Supercombinators en lambda-lifting (hoofdstuk 13).**
  - **Recursieve supercombinators (hoofdstuk 14).**
  - Fully-lazy lambda lifting (hoofdstuk 15).
  - SK combinatoren (hoofdstuk 16).
  - G-machine (hoofdstukken 18, 19 en 20).
- Geheugenmanagement en garbage collection (hoofdstuk 17).
- Tail call optimalisatie (hoofdstuk 21).
- Striktheidsanalyse (hoofdstuk 22).
- Typeclasses (niet in het boek).
- Input/output met monaden (niet in het boek).

Evaluatie

Ik zal een demonstratie geven van de opgeleverde features. De code zal worden ingediend met voorbeeldinput en documentatie.

Planning

Ik stel volgende persoonlijke deadlines.

21 februari: Verrijkte lambdacalculus omzetten naar lambdacalculus (hoofdstuk 6).

20 maart: Graafvoorstelling en graafreductie (hoofdstukken 10, 11 en 12).

3 april: Omzetting van Haskell naar verrijkte lambdacalculus (hoofdstukken 2 en 3).

10 april: Algebraïsche datastructuren en pattern matching (hoofdstukken 4 en 5).

24 april: Type checking en type inference (hoofdstukken 8 en 9).

1 mei: List comprehensions (hoofdstuk 7).

15 mei: Supercombinators en lambda-lifting (hoofdstuk 13).

29 mei: Recursieve supercombinators (hoofdstuk 14).