

# Verslag 2: Lazy evaluation door middel van graafreductie

Koen Pauwels

10 mei 2016

## 1 Inleiding

### 1.1 Waarom lazy evaluation?

In de meeste programmeertalen worden argumenten aan een functie geëvalueerd voordat de functie wordt opgeroepen. Dit noemen we een *call by value* strategie. Het is echter mogelijk dat een argument nooit wordt gebruikt in de functie. In dat geval wordt er overbodig werk geleverd.

We kunnen proberen dit overbodig werk te vermijden door de evaluatie van een argument uit te stellen totdat de waarde ervan nodig is. Dit noemen we een *call by need* of *lazy evaluation* strategie. Deze strategie is zo goed als uniek voor zuiver functionele talen. In imperatieve talen berust de betekenis van een programma op het in een correcte, voorspelbare volgorde plaatsvinden van bepaalde *side-effects*. Met *call by need* kan het erg moeilijk worden om uit te werken in welke volgorde de expressies zullen worden geëvalueerd, en dus in welke volgorde de side-effects zullen plaatsvinden.

### 1.2 Waarom een graafvoorstelling?

Wanneer een  $\lambda$ -expressie wordt ingeladen in de compiler, zal deze een boomvoorstelling van de expressie genereren. Zoals eerder vermeld worden argumenten aan functies enkel geëvalueerd wanneer de waarde nodig is. Naïeve normal-order reductie op de boomvoorstelling van de  $\lambda$ -expressie voldoet wel aan deze voorwaarde, maar zal vaak tot dubbel werk (en dus tot grote inefficiënties) leiden. Wanneer een parameter namelijk meerdere keren voorkomt in het lichaam van een expressie, zal elk voorval van die parameter vervangen moeten worden

door een kopie van het argument. Als we het programma voorstellen als een graaf in plaats van een boom, dan kunnen we elk voorval van de parameter in het lichaam van een  $\lambda$ -expressie vervangen door een *referentie* naar het argument, in plaats van een kopie ervan. Bij het evalueren van een argument zullen we ervoor zorgen dat de wortelnode van het argument wordt overschreven met het resultaat van de evaluatie. Elke referentie naar het argument verwijst dan automatisch en zonder extra kost naar het resultaat van de evaluatie.

TODO: uitleg adhv voorbeelden.

## 2 Programmarepresentatie

De graafvoorstelling moet alle constructies van de  $\lambda$ -calculus ondersteunen, maar verder verwachten we ook ondersteuning voor een aantal bijzondere builtin operaties en datastructuren. In theorie is de  $\lambda$ -calculus voldoende om alle programma's die we wensen uit te drukken, maar in praktijk willen we onder meer de machinevoorstelling gebruiken voor getallen, machine-operaties kunnen uitvoeren op die getallen, gestructureerde datatypes kunnen gebruiken, etc.

De basis van de graafvoorstelling is de *cel*, een eenvoudige datastructuur die uit drie elementen bestaat: een *tag* die aangeeft over welk type cel het gaat, en twee *fields*. Een field kan een pointer naar een andere cel bevatten (op deze manier worden de edges van de graaf geïmplementeerd), of een waarde waarvan de betekenis afhangt van het type cel en de context.

Listing 1: Definitie van de Cell datastructuur

```
typedef enum {VAR, APP, ABSTR, DATA, BUILTIN, CONSTR} Tag;
union Field {
    struct Cell* ptr;
    char* sym;
    int num;
    Builtin op;
    StructuredDataTag data_tag;
    StructuredDataPtr data_ptr;
};
struct Cell {
    Tag tag;
    union Field f1;
    union Field f2;
};
```

De eerste drie celtypes implementeren de structuren van de  $\lambda$ -calculus, zoals beschreven in *The Implementation of Functional Programming Languages (IFPL)*: variabelen, applicaties en abstracties.

- Een **VAR** cel gebruikt slechts 1 van zijn fields. Dat veld bevat een pointer naar een string die het symbool opslaat. Er is een efficiëntere implementatie mogelijk: we zouden bij het compileren elk symbool kunnen associëren met een uniek getal, en dit getal rechtstreeks in het veld opslaan. Ik heb besloten om de originele namen te behouden om debugging niet te bemoeilijken.
- De velden van een **APP** cel zijn eenvoudigweg celpointers naar de operator en de operand van een applicatie.
- Een **ABSTR** cel bevat een pointer naar het parameter symbool, en een celpointer naar het lichaam van de functie.

Het boek is minder helder over welke extra tags er nodig zijn om de implementatie te vervolledigen. Mijn implementatie maakt verder nog onderscheid tussen datacellen, builtins (operatoren en constanten) en constructors.

- **DATA** cellen worden gebruikt om gegevens in op te slaan. *IFPL* lijkt onderscheid te maken tussen cellen die getallen opslaan en cellen die lijsten (**CONS** cellen) opslaan, en eventueel nog aparte structured data cellen. Ik zag geen reden om de data cellen bewust te maken van de betekenis van de gegevens die ze dragen, aangezien dit steeds af te leiden valt uit de context. Een functie die iets met de data doet, bevat genoeg informatie om de data te interpreteren. **DATA** cellen worden dus zowel gebruikt om primitieve data in op te slaan, zoals integers, of om naar gestructureerde datastructuren te verwijzen. In het eerste geval bevat de datacel enkel de waarde van de primitieve data. In het tweede geval bevat de cel een *structured data tag* en een pointer naar de datastructuur. Wanneer de datastructuur werd aangemaakt door een constructor van een somtype, is het de structured data tag die onthoudt welke constructor dit was.
- Sommige operaties kunnen efficiënter worden uitgevoerd door een **BUILTIN** operatie te voorzien in de evaluator. Voor arithmetische en logische functies gebruiken we machine-instructies. Arithmetische en logische operatoren kunnen geïmplementeerd worden door middel van snelle machine-instructies. Conditionals en recursie kunnen ook worden uitgedrukt in  $\lambda$ -calculus, maar met ingebouwde **IF** en fixed point operatoren vermijden we heel wat overhead. Verder kan enkel de **SELECT** operator velden uit een gestructureerd datatype opvragen.

TODO: mss dit verplaatsen naar graafreductie section? Builtin operatoren verschillen ook van  $\lambda$ -abstracties in dat ze meer controle hebben over hoe hun argumenten gereduceerd dienen te worden. De `+` operator kan bijvoorbeeld niets zinnigs doen met niet-gereduceerde bomen, en zal dus steeds zijn argumenten volledig trachten te reduceren. De `IF` operator reduceert zijn predicaat eerst volledig, en daarna geeft het het correcte argument ongereduceerd terug.

Behalve operatoren zijn er ook enkele constanten die als builtins worden geïmplementeerd. Deze kunnen gezien worden als operatoren met 0 argumenten. Een voorbeeld hiervan is de `FAIL` constante, die gebruikt wordt om een gefaalde poging tot pattern matching te indiceren.

- Hoewel constructors in principe voorgesteld zouden kunnen worden door builtins, heb ik een apart `CONSTR` cetype gedefinieerd. Alle andere ingebouwde procedures hebben een vast aantal parameters per procedure: `+` heeft er steeds 2, `IF` heeft er 3, etc. Aangezien de gebruiker zijn eigen constructors kan definiëren met een arbitrair groot aantal velden, moet de `CONSTR` builtin omkunt met een variabel aantal argumenten. Het leek eenvoudiger om hiervoor een apart cetype te introduceren.

## 3 Implementatie van het reductiealgoritme

### 3.1 Selecteren van de volgende te reduceren expressie

We hebben al vastgesteld dat we een expressie pas wensen te evalueren wanneer haar waarde nodig is, maar we hebben nog geen procedure beschreven hoe we bepalen welke waarde we nodig hebben. Een expressie is reduceerbaar wanneer er zich ergens in de graaf een applicatie bevindt met in het linkerlid een  $\lambda$ -uitdrukking, of een applicatie van een builtin op het correcte aantal argumenten voor die builtin. Zo'n reduceerbare expressie noemen we een *redex*. Een strikt evaluatiealgoritme zal steeds alle redexes reduceren. Bij een luie strategie kan het reductieproces stoppen voordat alle redexes gereduceerd zijn.

