Practicum Programmeerprincipes 2006-2007

cderoove@vub.ac.be

REEKS 5
PAREN, LIJSTEN EN STROMEN IN PICO

Deze oefeningenreeks behandelt datastructuren die toelaten waarden te groeperen. Tot nu toe hebben we daarvoor steeds vertrouwd op de tabel-waarden van Pico. Nu we de basisprincipes van het programmeren in Pico onder de knie hebben, kunnen we echter ook zelf waardengroeperende datastructuren definiëren die na creatie zonder problemen een waarde meer of minder kunnen groeperen. Bij tabellen is dit niet mogelijk aangezien zij steeds hun initiële grootte behouden.

Een nieuwe datastructuur: het paar

Paren groeperen twee arbitraire waarden. In Pico kunnen deze eenvoudig voorgesteld worden als tabellen van grootte twee. Bij het werken met deze paren willen we echter denken in termen van paren en de elementen die ze groeperen en niet in termen van tabellen van grootte 2. Dit laat ons immers toe de interne voorstelling van paren te veranderen zonder vervolgens eveneens alle functies die op paren vertrouwen aan te moeten passen.

Oefening 1. Implementeer paren aan de hand van tabellen. Definieer hiervoor een constructor-functie `pair(first,second)` en vervolgens accessor-functies `first(pair)` en `second(pair)` die respectievelijk het eerste en tweede element van een paar gevormd door de functie pair teruggeven.

Schrijf vervolgens een functie `sumPair(pair)` die een paar neemt en de som van haar elementen teruggeeft.

Oefening 2. Herdefinieer de functies `pair`, `first` en `second` zodanig dat paren niet langer door tabellen, maar door functies voorgesteld worden. Een oproep van de functie `pair` geeft met andere woorden een functie in plaats van een tabel terug. Je kan je laten inspireren door de booleaanse waarden van Pico.

Ga na dat je functie `sumPair` zich in zijn originele vorm correct gedraagt op de door middel van functies voorgestelde paren.

Oefening 3. In de volgende oefeningen zullen we uit efficiëntieoverwegingen gebruik maken van paren geïmplementeerd aan de hand van tabellen. We zullen echter ook de hulpfuncties `is_atomic` en `is_pair` nodig hebben. Voorzie deze.

Een nieuwe datastructuur: de lijst

Lijsten groeperen een arbitrair aantal waarden. In tegenstelling tot tabellen zijn lijsten echter recursieve datastructuren in de zin dat een lijst een paar is waarvan het eerste element een arbitraire waarde is en het tweede element opnieuw een lijst is. De lege lijst kunnen we voorstellen aan de hand van `void`. De lijst `(1,2,3)` zou je dus kunnen opbouwen door de volgende expressie te evalueren:
`pair(1, pair(2, pair(3, void)))`

Oefening 4. Implementeer een functie `list` die een variabel aantal argumenten neemt en de lijst bestaande uit deze argumenten teruggeeft.

Oefening 5. Schrijf een recursieve functie `list_length` die een lijst neemt en er de lengte van teruggeeft.

Oefening 6. Schrijf een functie `append` die twee lijsten neemt en deze concateneert.

Oefening 7. Herimplementeer de functie `map(f,l)` zodat deze een functie toepast op elk element van een lijst en de resultaten verzamelt in een nieuwe lijst.

Oefening 8. Definieer een functie `filter(test, l)` die de elementen van de lijst `l` waarvoor de functie `test` naar `true` evalueert, verzamelt in een nieuwe lijst.

Oefening 9. Implementeer de functies `foldl(op,ini,lst)` en `foldr(op,ini,lst)` die respectievelijk een links-associatieve dan wel een rechts-associatieve operator `op` herhaaldelijk toepassen op alle elementen van een lijst `lst`, waarbij het eerste element van de lijst gecombineerd wordt met een eveneens meegegeven waarde `ini`.

```
foldl(+,0,list(1,2,3,4)) = ((((0 + 1) + 2) + 3) + 4)
foldr(+,0,list(1,2,3,4)) = (1 + (2 + (3 + (4+0))))
```

Oefening 10. Definieer een functie `copy(l)` die de lijst `l` kopieert aan de hand van de geschikte `fold` functie.

Herdefinieer vervolgens de functie `map(f,l)` aan de hand van de geschikte `fold` functie. Je kan hiervoor gebruik maken van een hulpfunctie `foldrf/foldlf` die equivalent is aan `foldr/foldf` op het gebruik van een functionele parameter na:

```
foldrf(op(x,y),ini,l) : foldr(op,ini,l)
foldlf(op(x,y),ini,l) : foldl(op,ini,l)
```

Oefening 11. Schrijf een functie `zip(l1,l2,null,op)` die twee lijsten samenritst met de operator `op`. De resulterende nieuwe lijst bevat de elementen die ontstaan door op paarsgewijs toe te passen op de elementen van `l1` en `l2`. Indien één van beide lijsten te kort is gebruik je `null` als neutraal element voor de operator. De expressie `zip(list(1,2,3,4),list(3,4),1,+)` zou bijvoorbeeld naar de lijst `<1+3,2+4,3+1,4+1>` evalueren.

Een nieuwe datastructuur: de stroom

Oefening 12. Kan je, analoog aan recursieve functies, met de huidige definitie van `pair`, `first` en `second` een paar construeren waarvan het tweede element opnieuw ditzelfde paar is? Pas je definities aan zodanig dat je wel degelijk een oneindige lijst of stroom eentjes `<1,1,1,1,1,1,1,.....>` kan definiëren:

```
> ones : ?????
> first(second(second(second(second(... second(ones)
... ))))) = 1
<function true>
```

Oefening 13. Schrijf een functie `nth(s,n)` die het n-de element uit een stroom teruggeeft zodanig dat je je definitie van `ones` kan testen:

```
> nth(ones,100) = 1
<function true>
```

Oefening 14. Implementeer een functie `integers_from(n)` die een stroom bestaande uit de gehele getallen vanaf n teruggeeft.

Oefening 15. Implementeer een functie `sum_streams(s1,s2)` die een stroom teruggeeft waarvan elk element de som van de overeenkomende elementen in de meegegeven stromen `s1` en `s2` is. Maak hiervoor gebruik van de eerder gedefinieerde functie `zip`!

```
> nth(sum_streams(integers_from(1),
                  integers_from(2)),3)=7
<function true>
```