Werkexemplaar

Nondeterminisme moet lazy zijn

Maarten Fokkinga, 10 dec 1985

Het redeneren over nondeterministische functionele programma's wordt aanzienlijk vereenvoudigd wanneer de lazyness van de evaluatie geen implementatie-vrijheid is maar daadwerkelijk tot de semantiek behoort.

* * *

Er zijn velerlei redenenen om nondeterministische constructies ook in functionele talen op te nemen. Bijvoorbeeld, nondeterministische guarded expressions (om overspecificatie te voorkomen), nondeterministische fair merging (om reactive systems zoals operating systems te kunnen programmeren, maar ook om op natuurlijke wijze verzamelingen te kunnen representeren, zie [Fokkinga 1985]), en de nondeterministische bottom-avoiding choice (waarvoor ik nog geen dringende reden weet aan te voeren, maar waarmee guarded expr's zijn uit te drukken).

In een eerder verhaal, [Fokkinga 1985], heb ik het vermoeden geuit dat er voor nondeterministische expressies een zinvolle semantische gelijkwaardigheidsrelatie = en een semantische verfijningsrelatie >> is te definieren, zó dat ~~bijvoorbeeld~~ = voldoet aan

aan "de wetten voor de gelijkheid", te weten de reflexiviteit, symmetrie, transitiviteit, congruentie en substitutiviteit. De 2 laatste van deze ~~twee~~ wetten zijn:

(congr) $e = e' \Rightarrow C[e] = C[e']$

(subst) $e = e' \Rightarrow (P(e) \Leftrightarrow P(e'))$

(voor een nader te bepalen klasse van predicaten P). Ruwweg gezegd is $e = e'$ gedefinieerd als: hetzij e en e' bevatten géén nondeterministische operator of functie en zijn beide semantisch equivalent in de gewone zin, hetzij of geldt voor iedere berekening van de een, met resultaat r, dat er een berekening van de ander is met een resultaat r' met $r = r'$.

Helaas hebben we aan deze wetten nog niet genoeg om gemakkelijk over nondeterministische programma's te kunnen redeneren. Allereerst geldt dat de gelijkheidstest (als operator uit de programmeertaal) nogal verschilt van de semantische gelijkwaardigheid: de gelijkheidstest evalueert beide operanden volgens één van de vele mogelijkheden en vergelijkt de aldus gevonden resultaten, terwijl de semantische gelijkwaardigheid over alle mogelijke evaluaties van de operanden een uitspraak doet. We noteren de gelijkheidstest met == (een "sterker" teken omdat het een "sterkere" relatie is). Een voor-

beeld is dan: $(0 \,\square\, 1) = (0 \,\square\, 1)$ maar $((0 \,\square\, 1) == (0 \,\square\, 1)) \neq$ true
Dit verschil tussen de gelijkheidstest en de gelijkwaardigheid is niet zó bezwaarlijk; ook zonder nondeterminisme is er al verschil (bijvoorbeeld doordat de test niet termineert voor sommige wel gelijkwaardige expressies en doordat er zowiezo geen correcte en complete test voor functionele waarden kan bestaan).

Een tweede, veel grotere moeilijkheid, is de volgende; en daar zullen we het verder alleen maar over hebben. De bewijsregel

$$\frac{P(e),\quad P(e')}{P(e \,\square\, e')} \ (\square\, funct)$$

lijkt niet algemeen geldig. Bijvoorbeeld, definieer

$$f\ x = (x + x == 2*x)$$

en beschouw dan $f\ x = true$: er geldt wel $P(0)$, i.e. $f\ 0 = true$, en ook $P(1)$, nml. $f\ 1 = true$, maar niet $P(0 \,\square\, 1)$, want

$$\begin{aligned} f(0 \,\square\, 1) &= (0 \,\square\, 1) + (0 \,\square\, 1) == 2*(0 \,\square\, 1) \\ &= 0+0 == 2*0 \,\square\, 0+1 == 2*0 \,\square\, \ldots \,\square\, 1+1 == 2*1 \\ &= true \,\square\, false \,\square\, \ldots \,\square\, false \,\square\, true \\ &\neq true \end{aligned}$$

Het falen van regel ($\square$ funct) is des te verrassender omdat de "overeenkomstige" regel voor imperatieve talen wel geldig is:

$$\frac{\{P\}\ S_1\ \{Q\},\quad \{P\}\ S_2\ \{Q\}}{\{P\}\ S_1 \,\square\, S_2\ \{Q\}} \ (\square\, imp)$$

Inderdaad, nondeterminisme is al gemeengoed in imperatieve talen en met name de guarded commands zijn een plezierige constructie om algoritmen elegant te formuleren. Het redeneren over nondeterministische imp. programma's is niet moeilijker dan over deterministische programma's.

Waarom, dan, faalt regel ($\square$ funct) en slaagt regel ($\square$ imp)? De reden lijkt te zijn dat de normal form reductie-strategie (ook wel outside-in strategie genoemd) de boosdoener is. Daardoor immers kan er bij de berekening van $f\ x = (x + x == 2 \cdot x)$ op drie plaatsen opnieuw gekozen worden als er een keuze-mogelijkheid in het argument voor x aanwezig is. Imperatieve talen worden volgens de applicative strategie (ook wel inside-out of compositionele strategie, in Structuur van Programmeertalen)) geevalueerd. En volgens de inside-out strategie is $f(0 \,\square\, 1) = f\,0 \,\square\, f\,1 = (0+0 == 2*0) \,\square\, (1+1 == 2*1) = true$

We zouden hieruit kunnen concluderen dat de outside-in strategie weliswaar de korrektheids beschouwingen voor deterministische (funktionele) programma's vereenvoudigt, (~~maar~~ zoals in de literatuur al vele malen beargumenteerd is, zie bijvoorbeeld [Turner 1982]), maar voor nondeterministische programma's juist bemoeilijkt. Het zou ongetwijfeld jammer zijn als we alleen daarom weer terug moeten gaan naar het inside-out tijdperk. Is er geen strategie die de goede eigenschappen van beide verenigt?

Het antwoord is "Ja: lazy evaluation!". De reden is dat, ruwweg gezegd, argumenten hoogstens eenmaal gereduceerd worden --zoals bij de inside-out strategie-- alhoewel het tijdstip daarvan bepaald wordt zoals bij de outside-in strategie. (Om precies deze reden combineert lazy evaluation ook al de goede tijds-efficientie-eigenschappen van call-by-name en call-by-value!) Een eenmaal gedane keus wordt dus niet meer opnieuw gedaan maar is ook geldig voor alle voorkomens uit dezelfde "kloon". Voor ons voorbeeld vinden we onder lazy evaluation dat

$$
\begin{aligned}
f(0 \square 1) &= {}^{\alpha}(0 \square 1) + \alpha == 2*\alpha \\
&= ({}^{\alpha}0 + \alpha == 2*\alpha) \square ({}^{\alpha}1 + \alpha == 2*\alpha) \\
&= \text{true} \square \text{true}
\end{aligned}
$$

$$= \text{true}$$

We hebben hier de notatie $\ldots\alpha\ldots{}^{\alpha}(\ldots)\ldots\alpha\ldots\alpha\ldots$ gebruikt om sharing van expressies aan te geven: alle α's staan voor eenzelfde expressie, namelijk degene die met α-als-prefix-superscript benoemd is. (Alle expressies α vormen te samen een "kloon"; zij zijn uit eenzelfde voorkomen afkomstig.) Deze notatie heeft al eerder zijn vruchten afgeworpen, zie [Fokkinga 1985 b].

<u>Vermoeden</u>

De regels

(NB. bij bottom-avoiding choice $\Rightarrow$ ipv $\Leftrightarrow$)

$$
\begin{array}{ll}
(\square\text{funct}) & P(e) \;\&\; P(e') \Leftrightarrow P(e \square e') \\
(\square\text{distrib.}) & C(e) \square C(e') = C(e \square e')
\end{array}
$$

zijn universeel geldig als we voor predicaten en contexten ook het sharing-principe van lazy evaluation toepassen, dwz. een definitie zoals

$$
\begin{aligned}
P(x) &=_{def} x + x = 2*x \\
C(x) &=_{def} x + x
\end{aligned}
$$

wordt geinterpreteerd als

$$P(x) =_{def} {}^{\alpha}x + \alpha = 2*\alpha$$

$$C(x) =_{def} {}^{\alpha}x + \alpha$$

□

We moeten dus wel erg precies zijn in het aangeven welke expressies wél en welke niet geshared worden. Dit lijkt mij niet bezwaarlijk: voor sharing van niet-elementaire expressies is de sharing-notatie alleen maar prettig (het beoogt je van schrijfwerk zonder de leesbaarheid geweld aan te doen), en voor elementaire expressies, variabelen dus, lijkt het zinvol om af te spreken dat alle voorkomens van eenzelfde variabele ge-shared worden (en voorkomens van verschillende variabelen niet). Bijvoorbeeld, beschouw nogmaals de beweringen

$$x + x = 2 * x$$
$${}^{\alpha}(0 \square 1) + \alpha = 2 * \alpha$$
$$(0 \square 1) + (0 \square 1) = 2 * (0 \square 1)$$

Alhoewel de derde bewering niet waar is, is de eerste dat wel: variabelen zijn by default geshared. De tweede bewering is dus een substitutie-instantie van de eerste; de derde niet!

Het feit dat alle voorkomens van eenzelfde variabele (of preciezer: alle voorkomens die door eenzelfde binding gebonden worden) geshared zijn, betekent dat in iedere context iedere variabele maar één waarde aanduidt (zij het dat die waarde nondeterministisch bepaald kan zijn). [Clinger 1982] noemt de semantiek dan ook "singular"; (hij onderscheidt 8 mogelijke semantieken!). De term "environmental transparency" is een alternatief.

Er zijn vele pogingen gedaan (en geslaagd) om een denotationele semantiek te geven voor (al of niet functionele) talen met nondeterminisme. Zie bijvoorbeeld [Clinger 1982] en de daarin genoemde verwijzingen. Maar naar mijn weten is er voor functionele talen met lazy evaluation nog geen onderzoek geweest (met publicatie) naar de geschiktste vorm van nondeterminisme (t.a.v. praktisch nut tesamen met hanteerbaarheid voor korrektheidsbewijzen).

Literatuur

Clinger, W., Nondeterministic call by need is neither lazy nor by name. In Conf Record of the 1982 ACM Symp on LISP and Functional Programming, 1982, pp 226-234.

Fokkinga, MM, Fair nondeterministic choice considered necessary. Notitie, 17 november 1985.

Fokkinga, M.M., Lazy evaluation op expressie-nivo uitgelegd. Notitie, (herzien) 3 dec 1985.

Turner, D.A., Functional programming and proofs of program correctness. In Tools and Notions for Program Constructions (ed D. Neel), Cambridge University Press, Cambridge, UK, 1982, pp 187-210.

Aanhangsel: guarded expressions

Ik doe hier een vluchtige poging een bewijsregel voor guarded expressions op te stellen (voor lazy geëvalueerde expressions! dus met ~~referential~~ environmental transparency). Hier is de regel.

$$\frac{e_1 \vee e_2 \not\Rightarrow \underline{true} \qquad e_1 \gg \underline{true} \supset P(e_1') \quad \& \quad e_2 \gg \underline{true} \supset P(e_2')}{P((e_1 \to e_1' \;[\!]\; e_2 \to e_2'))}$$

Let erop dat sharing plaats vindt in de eerste premisse en, los daarvan, ook in de gehele tweede premisse: een keus in de ene guard kan, als er sharing is met de andere guard, de uitkomst daarvan mede bepalen. Premisse (i) eist dat hoe-dan-ook elke mogelijke berekening van $e_1 \vee e_2$ tot $\underline{true}$ evalueert. De conditie $e_1 \gg \underline{true}$ zegt "Er is een evaluatie van e_1 tot $\underline{true}$" en sluit niet uit dat e_1 ook eut tot $\underline{false}$ evalueert; (divergentie kan ~~ook~~ niet vanwege premisse (i)).

Hiermee is $P(x, y, \max\ x\ y)$ te bewijzen, (dus geldt ook $P({}^{\alpha}(0 \;[\!]\; 1), {}^{\beta}(0 \;[\!]\; 1), \max\ \alpha\ \beta)$), met

$$\max\ x\ y \;=\; (x \le y \to y \;[\!]\; y \le x \to x)$$
$$P(x, y, z) \;=\; (x \le z \wedge y \le z \wedge (x = z \vee y = z)) = \underline{true}$$