Een codegenerator voor het opsommen van programma's

Gilles Coremans
Promotor: Tom Schrijvers
KU Leuven

Abstract

Het is van groot belang dat implementaties van programmeertalen voldoen aan correctheidseigenschappen over hun evaluatie en uitvoering. Nagaan of deze eigenschappen kloppen is echter niet vanzelfsprekend; naast het formuleren van eigenschappen is er nog de kwestie van geschikte programma's te vinden om deze eigenschappen op te testen.

We stellen een uitbreiding voor van de *Autbound*-codegenerator die efficiënt willekeurige programma's genereert aan de hand van enumeraties. Met deze enumeraties kunnen we testen of implementaties van programmeertalen aan eigenschappen voldoen. We demonstreren het nut hiervan door eigenschappen van System F, F_i^+ en F_{co} te testen en bugs te vinden.

1 Inleiding

1.1 System F

System F is een polymorf getypeerde lambdacalculus [8] die onder meer gebruikt wordt als de basis van Haskell.

"Getypeerd" betekent dat met termen types worden geassocieerd. Deze types bepalen en garanderen de in- en uitvoerwaardes van functies. Termen waarvoor we een type kunnen vinden zullen we *goedgetypeerd* noemen. Het is echter niet mogelijk om aan alle termen een type toe te wijzen.

Met "polymorf getypeerd" bedoelen we dat types universeel gekwantificeerde variabelen kunnen bevatten. Dit maakt het systeem veel krachtiger, en zorgt ervoor dat we functies kunnen opstellen die voor alle types werken. Een voorbeeld is de identiteitsfunctie, die als type $\forall\chi : \chi \to \chi$ heeft:

$$\Lambda\chi : \lambda x^{\chi} : x$$

System F introduceert twee elementen: de type-abstractie Λ die een nieuwe typevariabele introduceert, en de type-applicatie die een universele kwantificatie invult met een concreet type. We annoteren ook iedere variabelebinding met een type om uitdrukkingen te helpen typeren.

1.2 Correctheidseigenschappen

Een praktische implementatie van een programmeertaal moet aan een aantal eigenschappen voldoen om bruikbaar te zijn.

Een voorbeeld is de stelling van Church-Rosser: als een term een normaalvorm heeft, is deze uniek [3]. Deze eigenschap geldt in zowel de gewone lambdacalculus als in System F. Praktisch gezien betekent dit dat de volgorde van bèta-reducties niet uitmaakt, wat verschillende evaluatiestrategieën mogelijk maakt.

Haskell maakt hiervan gebruik in de vorm van haar typische *lazy* karakter. Aangezien de reductieorde niet uitmaakt, kan Haskell de evaluatie van een uitdrukking uitstellen tot wanneer de waarde nodig is, wat veel nieuwe programmeertechnieken en optimalisaties toelaat.

1.3 Property-based testing

Waar het vaak al veel werk vereist om eigenschappen van een theoretisch systeem te bewijzen, is het ontwerpen van een correcte implementatie ook moeilijk. Zelfs in implementaties van systemen die een eigenschap bezitten is het niet gegarandeerd dat de implementatie die eigenschap ook bezit. Ook is het in het algemeen niet mogelijk om eigenschappen automatisch te bewijzen. Toch is het vaak nuttig om bugs in een implementatie te proberen detecteren door middel van tests.

Hiervoor dient *property-based testing*, een techniek die ontworpen is om eigenschappen te testen. In Haskell bestaan er verschillende libraries die deze techniek implementeren, waaronder QuickCheck [4], SmallCheck [9] en FEAT [5]. Deze libraries laten de programmeur toe om een eigenschap in Haskell te formuleren, die dan automatisch getest wordt op een groot aantal invoerwaarden. Voor het automatisch testen van programmeertalen is het echter moeilijk om efficiënt vele willekeurige programma's te genereren.

1.4 *Autbound*

Deze paper beschrijft een uitbreiding aan de bestaande *Autbound*-codegenerator, met als doel automatisch programma's te genereren in talen die beschreven zijn in de *Autbound*-specificatietaal [1]. Deze codegenerator dient voor het automatisch manipuleren van *abstract syntax trees*, boomvoorstellingen van programma's. In het bijzonder betreft de code gegenereerd door *Autbound* de variabelenbinding in programma's. De reden dat hiervoor een codegenerator gebruikt wordt is dat deze code vaak uitgebreid en fragiel is, ondanks het feit dat code voor variabelenbinding meestal zeer gelijkaardig is tussen verschillende systemen.

Een ander voordeel van *Autbound* tegenover andere gelijkaardige codegeneratoren is dat het ook de Bruijn-notatie on-

```
1  data Comble where
2    Empty :: Comble e
3    Pure  :: e → Comble e
4    Alt   :: Comble e → Comble e → Integer →
         Comble e
5    App   :: Comble (arg → e) → Comble arg →
         Integer → Comble e
6
7  instance Applicative Comble where
8    pure x  = Pure x
9    f <*> x = App f x (card f * card x)
10
11 instance Alternative Comble where
12   empty   = Empty
13   t <|> u = Alt t u (card t + card u)
14
15 (!) :: Comble e → Integer → e
16 Pure x     ! 0  = x
17 Alt t u _ ! k  = if k < m
18                  then t ! k
19                  else u ! (k − m)
20   where m = card t
21 App t u _ ! k  = (t ! (k `div` n))
22                $ (u ! (k `mod` n))
23   where n = card u
24
25 card :: Comble e → Integer
26 card Empty       = 0
27 card (Pure _)    = 1
28 card (Alt _ _ n) = n
29 card (App _ _ n) = n
```

Listing 1: De Haskell-definitie van Combles en functies om Combles te indexeren.

dersteunt. Dit is een notatie voor lambdacalculi waarbij variabelen genoteerd worden als cijfers die aanduiden bij welke binder ze horen in plaats van als een naam [6]. Op deze manier vermijdt men het probleem van *name collision*.

2 Probleemstelling en aanpak

We willen automatisch eigenschappen kunnen testen voor talen gespecificeerd met *Autbound*. Hiervoor moeten we echter automatisch programma's kunnen genereren, en dit leidt ons tot de volgende onderzoeksvraag:

> Is het mogelijk om automatisch willekeurige programma's te genereren voor programmeertalen gespecificeerd in de *Autbound*-specificatietaal?

Dit moet efficiënt gebeuren en programma's van een redelijke grootte opleveren. De generator moet ook praktisch bruikbaar zijn voor het automatisch testen van eigenschappen.

We zullen de onderzoeksvraag positief beantwoorden en het nut ervan demonstreren door enkele implementaties van programmeertalen te testen en bugs te vinden.

We realiseren onze bijdrage als een uitbreiding aan de *Autbound*-codegenerator, zodat we implementaties gebaseerd op deze codegenerator kunnen testen. We gebruiken geen rechtstreekse generatie van willekeurige programma's, maar stellen enumeraties op van alle programma's van een bepaalde grootte. Hiervoor passen we een bestaande datastructuur, een *Comble*, toe. Onze bijdrage bestaat uit:

- Het automatisch opstellen van Combles op basis van een specificatie. (Sectie 4.1)
- De behandeling van complexe variabelenbinding bij Combles. (Sectie 4.2)
- Het opstellen van Combles van datastructuren bestaande uit meerdere datatypes. (Sectie 4.3)
- De toepassing van Combles als invoer voor *property-based testing*: (Sectie 5)
  - Het gebruik van Combles om bugs te vinden en te helpen oplossen.
  - Een bespreking van de beperkingen van onze aanpak, met name een kleine fractie "interessante" termen en een hoog geheugengebruik.

3 Enumeraties specificeren met *Combles*

3.1 Enumeraties

De keuze voor enumeratie in plaats van random generatie is weloverwogen. Aangezien elk programma een welbepaalde index in de enumeratie heeft en de enumeratie een welbepaalde kardinaliteit heeft, kan men een random generator verkrijgen door simpelweg willekeurige indices in de enumeratie te kiezen. Het omgekeerde, het creëren van een enumeratie uit een random generator, is in het algemeen niet mogelijk.

Bovendien kunnen enumeraties gebruikt worden om tests uit te voeren die niet mogelijk zijn met random generatie. Zo kan men met een enumeratie die in stijgende grootte van termen geordend is een eigenschap op alle termen tot een bepaalde grootte testen. Als hierbij geen bugs gevonden worden, hebben we een ondergrens aan de grootte en dus complexiteit van programma's die de eigenschappen schenden, wat in veel gevallen nuttiger is dan willekeurige tests van oplopende groottes. Deze redenen motiveerden ook de ontwikkeling van enkele property-based testing libraries gebaseerd op enumeraties, zoals SmallCheck [9].

Aangezien Haskell *lazy* is, is het niet nodig om de enumeratie expliciet te genereren. Het volstaat om een geschikte datastructuur te definiëren om de enumeratie voor te stellen en een functie te schrijven die de enumeratie recursief specificeert.

3.2 Combles specificeren

Concreet gebruiken we een datastructuur uit nog ongepubliceerd werk van Ralf Hinze, genaamd *Comble*. Deze datastructuur wordt in Haskell gedefinieerd zoals in de eerste 13 regels van Listing 1.

De constructoren van de Comble zijn syntactische implementaties van de *Applicative* en *Alternative* typeclasses. In dit geval dienen ze geïnterpreteerd te worden in de context van recursieve datastructuren, zoals de syntaxboom van een programma. De <|>-combinator van de Alternative typeclass stelt hier een keuze voor tussen twee verschillende Combles van hetzelfde type. Door deze combinator te nesten kan men een keuze tussen een willekeurig aantal Combles voorstellen. Met de `pure`-functie kunnen we een comble construeren die slechts één waarde bevat, namelijk het argument van `pure`. Met enkel deze twee functies zou men echter enkel de keuze tussen geïnstantieerde waardes kunnen uitdrukken.

De <*>-combinator van de Applicative typeclass speelt hier dus een essentiële rol. Deze combinator laat toe om een Comble van functies toe te passen op een Comble van argumenten [7]. Dit kan men ook als een soort keuze zien, namelijk de keuze in welke argumenten we op welke functie toepassen. Op deze manier kunnen we Combles gebruiken als het argument van een constructor, om zo recursieve datastructuren recursief op te bouwen.

De laatste functie, `empty`, stelt een onmogelijke keuze voor, die nooit gekozen kan worden, of een Comble die geen enkele waarde bevat. Men kan dit gebruiken om onmogelijke constructies te verbieden. Als men bijvoorbeeld een variabele wilt genereren, maar er zijn geen binders in scope, kan men zo een lege Comble als resultaat hebben.

De enumeratie stelt dus een opeenvolging van keuzes tussen verschillende constructoren voor. Men kan dit ook beschouwen als een trie, waar de nodes geen strings maar constructoren bevatten. Breng hierbij wel het feit in rekening dat recursieve datastructuren vaak zelf een boomstructuur hebben, waardoor men een "prefix" ruimer moet opvatten dan bij strings.

Neem als voorbeeld een kleine aritmetische "taal", `Ari`, die bestaat uit letterlijke getallen, een functie `Sum` die de som van twee getallen berekent, en een functie `Pred` die 1 aftrekt van een getal. Uitgedrukt als een recursieve datastructuur in Haskell wordt dit:

```
data Ari = Lit Integer
         | Pred Ari
         | Sum Ari Ari
```

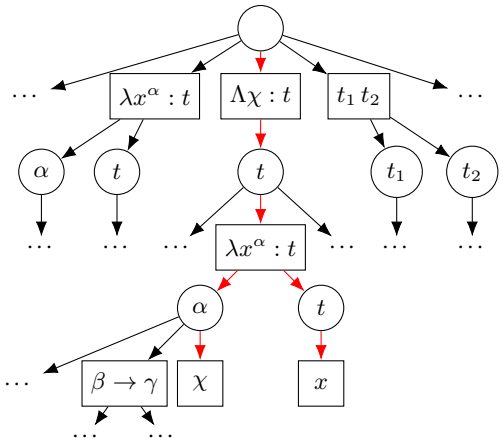
Een Comble opstellen voor deze datastructuur betekent het opstellen van een recursieve functie die deze Comble genereert. Deze functie heeft één argument, de grootte van de termen in de Comble. We definiëren de grootte van een term als het aantal constructoren in die term.

Als we een functie `genInt` hebben die een Comble van getallen genereert ziet de code voor deze Comble er zo uit:

```
genAri 0 = empty
genAri 1 = Lit <$> genInt
genAri n = Pred <$> genAri (n-1)
       <|> asum [Sum <$> genAri n1 <*> genAri
            n2 | (n1, n2) <- split2 (n-1)]
```

Het basisgeval van deze recursieve functie is de constructor `Lit`, aangezien deze zelf geen argument van type `Ari` heeft en dus steeds grootte 1 heeft. Een keuze tussen de twee andere constructoren, `Pred` en `Sum`, vormt het recursieve geval van de functie. `Pred` neemt slechts één argument, van type `Ari`. Het argument voor deze constructor is dus onze functie met een grootte-parameter verminderd met 1.

Het geval van `Sum` is complexer. Deze constructor heeft twee argumenten van type `Ari`. Indien we twee keer de grootte-parameter verminderd met 1 doorgeven, genereren we geen termen van exact de gegeven grootte. We moeten de grootte-parameter dus "verdelen" over de twee argumenten. De <*>-combinator laat echter geen verbanden tussen haar argumenten toe. Om dit op te lossen construeren we met <|> een keuze tussen alle mogelijke verdelingen van de grootte over de argumenten. De functie die alle tupels genereert die als som n hebben zullen we `split2` noemen. We gebruiken hiervoor een list comprehension en een functie, `asum`, die een keuze opstelt tussen alle elementen van een lijst.



Figuur 1: Versimpelde voorstelling van de constructie van de System F term $\Lambda\chi : \lambda x^{\chi} : x$ uit een Comble.

We hebben echter nog steeds een probleem: indien de grootte-parameter 2 is, moet men voor `Sum` een grootte van 1 verdelen over twee parameters, wat niet gaat. De oplossing hiervoor is om simpelweg een tweede base case toe te voegen. De base case voor grootte 1 bestaat enkel uit `Lit`, maar we voegen nu een tweede base case, voor grootte 0, toe. Deze is `empty`, een onmogelijke keuze.

3.3 Waardes uit Combles berekenen

Natuurlijk is het ook belangrijk om efficiënt waardes uit een Comble te kunnen berekenen. De functie voor indexatie is beschreven in Listing 1 vanaf regel 15. Hierin schuilt het belangrijkste detail van Combles: als de kardinaliteit van subCombles gekend is, kan men zeer snel bepalen of een index tot die Comble behoort of niet. Op deze manier kunnen we snel waardes uit Combles halen. De kardinaliteit van een Comble is makkelijk te beschrijven. De kardinaliteit van `Pure` is 1 en de kardinaliteit van `Empty` is 0. Dan kan men inductief de kardinaliteit van subbomen bepalen: de kardinaliteit van `Alt`, een keuze, is de som van de kardinaliteiten van keuzes, en de kardinaliteit van `App` is het product van de kardinaliteiten, aangezien elke functie in de Comble van functies toegepast wordt op elke waarde in de Comble van argumenten.

Ook het gebruik hiervan om waardes te vinden is vrij eenvoudig. Als een enumeratie bestaat uit een `Alt` en we willen de waarde op positie n vinden, dan zal deze aanwezig zijn in de linker-Comble indien n kleiner is dan de kardinaliteit van de linker-boom. Anders zal deze in de rechterboom te vinden zijn. Voor `App` is de situatie complexer, aangezien de waarde bestaat uit een functie uit de linkerboom toegepast op een waarde uit de rechterboom. We berekenen hier welke combinatie bij de index hoort door voor de index in de linkerboom de index te delen door de kardinaliteit van de rechterboom en naar beneden af te ronden, en voor de index in de rechterboom de index modulo de kardinaliteit van de rechterboom te nemen.

Deze recursieve definitie van kardinaliteiten betekent wel dat als men een waarde op index n wil berekenen, men eerst

elke node in de Comble met index kleiner dan n moet bezoeken. Dit zorgt ervoor dat het zeer inefficiënt is om één enkele waarde uit de Comble te halen. Combles zijn echter wel zeer efficiënt te gebruiken voor veel opvragingen in eenzelfde Comble, of om een oplopende enumeratie op te stellen, aangezien de kardinaliteiten hergebruikt kunnen worden voor volgende berekeningen.

4 Automatisch programma's opsommen

4.1 Combles genereren op basis van specificaties

De *Autbound*-specificatie bevat een lijst met datatypes, hun constructoren en de argumenten van die constructoren. *Autbound* maakt een onderscheid tussen drie soorten constructoren. Gewone constructoren binden geen variabelen en verwijzen ook niet naar variabelen. Deze bevatten geen belangrijke extra informatie. Binders introduceren een nieuwe variabele, met informatie over welk soort variabele dit is en in welke argumenten deze gebonden is. Ten laatste zijn er variabeleconstructoren, die informatie bevatten over naar welk soort variabele ze verwijzen. Een volledige beschrijving van de specificatie is te vinden in [1].

Listing 2 is een voorbeeld van een specificatie voor de gewone lambdacalculus. `TmApp` is een gewone constructor met twee argumenten, beide van datatype `Term`. `TmAbs` is een binder, het introduceert een nieuwe termvariabele, `z`. De variabele wordt in het enige argument `x` gebonden in de context `ctx`. `TmVar` is een variabeleconstructor, deze verwijst naar een variabele in de context `ctx`.

In Listing 3 zien we de resulterende code. De functie `termOfSize` bepaalt een Comble voor lambdatermen. We zien dat we één constructor van grootte 1 hebben, `TmVar`, die als argument een Comble van mogelijke variabelen neemt. Voor grootte `n` hebben we twee constructoren, `TmAbs` en `TmApp`, waartussen we kiezen met `<|>`. Voor `TmApp` moeten we de grootte verdelen aangezien deze twee argumenten heeft. De uitdrukking voor `TmAbs` is echter vrij simpel.

De geïmplementeerde codegenerator bestaat uit ongeveer 150 lijnen Haskell-code, en kan men vinden op https://gitlab.ulyssis.org/operand/asttool.

4.2 Variabelebinding in Combles

Voor het genereren van programma's met correct gebonden variabelen moeten we meer informatie bijhouden dan enkel de gewenste groottes. We moeten ook de variabelen bijhouden die op elk punt in de Comble gebonden zijn. Dit doen we met een extra functieparameter. Deze parameter is een recursief datatype, een lijststructuur die het type en de volgorde van de binders die we gepasseerd zijn bijhoudt.

Dit laat ons toe om meerdere "contexten" van variabelen, bijvoorbeeld term- en typevariabelen, in eenzelfde datastructuur bij te houden. We kunnen dit ook gebruiken om termen met vrije variabelen te genereren indien dit gewenst is.

4.3 Datatypes

Bij de groottes van de datatypes die we niet rechtstreeks genereren, maar die voorkomen als argumenten van de constructoren van het hoofddatatype moet men ook een keuze maken.

```
namespace TermVar: Term

sort Term
    inh ctx  TermVar
    | TmVar (x@ctx)
    | TmAbs (x: Term) [z: TermVar]
        x.ctx = lhs.ctx, z
    | TmApp (t1: Term) (t2: Term)
```

Listing 2: *Autbound*-specificatie van de lambdacalculus.

```
termOfSize env 0      = empty
termOfSize env 1      = TmVar <$> (getTermVar env (Z))
termOfSize env nTerm =
    (TmAbs <$> (termOfSize (STermVar env) (nTerm − 1)))
  <|> (asum (map (\n → (TmApp <$> (termOfSize env (n !! 0)) <*>
       (termOfSize env (n !! 1)))) (splitn 2 (nTerm − 1))))

getTermVar (Z)              _   = empty
getTermVar (STermVar next) abs = (pure abs)
                                 <|> (getTermVar next (STermVar abs))
```

Listing 3: De gegenereerde Comble-code voor Listing 2.

Men kan de totale grootte van subdatatypes vastleggen, waardoor men precies weet hoeveel constructoren er in alle gegenereerde termen zullen voorkomen. Deze aanpak heeft echter het probleem dat sommige programma's veel meer gebruik zullen maken van subdatatypes dan anderen. Bijvoorbeeld, de volgende System F term, bestaande uit vijf constructoren, heeft maar op één plaats een type, namelijk in het type van de variabele x:

$$\Lambda\chi : \Lambda\gamma : \Lambda\rho : \lambda x^{\chi} : x$$

Terwijl de volgende term, ook van grootte vijf, drie types nodig heeft:

$$\Lambda\chi : \lambda x^{\chi} : \lambda y^{\chi} : \lambda z^{\chi} : x$$

Bij deze aanpak is het dus moeilijk om alle termen van een bepaalde grootte te genereren zonder dat bepaalde termen types hebben die veel groter zijn dan gewenst.

Het alternatief is om op elke plaats waar een subdatatype nodig is een subdatatype van de gevraagde grootte te genereren, in plaats van deze grootte te proberen verdelen. Het is duidelijk dat op deze manier niet alle termen mogelijk zijn, bijvoorbeeld:

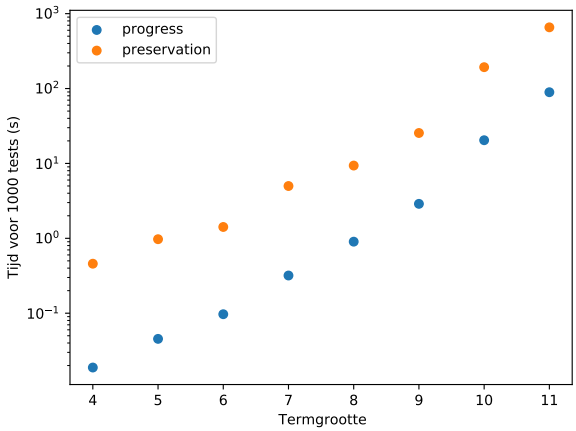
$$\Lambda\chi : \lambda x^{\chi} : \lambda y^{\forall\gamma:\chi} : y$$

Deze term heeft een type van grootte 1 en een type van grootte 2. De oplossing is om datatypes te genereren met een grootte kleiner dan of gelijk aan de gevraagde grootte.

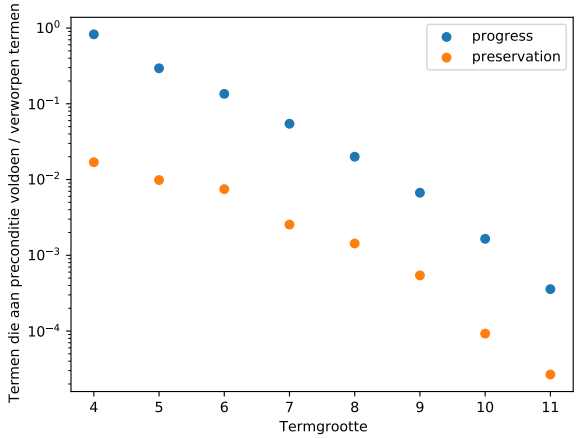
4.4 Correctheid

Het is natuurlijk belangrijk dat de code die invoer genereert voor tests zelf correct is. Dit is echter moeilijk om te garanderen. Voor kleine groottes kan men de enumeraties manueel controleren op alle verwachte termen, maar hier wordt de kardinaliteit zeer snel te groot voor.

Onze implementatie heeft als voordeel dat het een codegenerator is, wat betekent dat men fouten in de relatief beperkte generatorcode kan terugvinden en oplossen. Ook betekenen fouten niet dat de tests volledig nutteloos zijn: foute code zal niet alle bugs vinden, maar heeft nog steeds een grote kans om sommige bugs te vinden. Als men echter een foutief tegenvoorbeeld vindt, zal men dit natuurlijk in detail bestuderen om de bug op te lossen en op deze manier zien dat het tegenvoorbeeld slechtgevormd is.



Figuur 2: De tijd nodig om 1000 tests voor een eigenschap van System F uit te voeren in functie van de grootte van de gegenereerde termen.



Figuur 3: De fractie van van termen die aan de preconditie van een eigenschap van System F voldoen in functie van de grootte van de gegenereerde termen.

5 Toepassing: implementaties testen

Om onze bijdrage te evalueren hebben we eigenschappen van enkele bestaande implementaties getest. De geteste systemen zijn een implementatie van System F [8] en van de gerelateerde systemen F_i^+ en F_{co} [2]. Voor deze systemen bestaat al een Autbound-specificatie, wat het testen vergemakkelijkt. De specificaties en implementaties kan men vinden op https://gitlab.ulyssis.org/operand/asttool.

Het is echter niet mogelijk gebleken om eigenschappen exhaustief te testen. De reden hiervoor is dat de set van termen van een bepaalde grootte voor nuttige groottes simpelweg te groot is. Er zijn al 11466159314588 ($\approx 10^{13}$) verschillende termen in de enumeratie van de vrij beperkte System F-termen van grootte 7 met types tot grootte 5. Dit probleem wordt steeds erger naarmate men grotere termen en types genereert.

Daarom, en omdat deze library zeer wijdverspreid is, hebben we onze tests uitgevoerd met de QuickCheck property-based testing library. We genereren ook functies die grootteparameters en een QuickCheck `Property` aannemen en deze property testen met termen uit de enumeratie.

We gebruiken deze functies in plaats van de datatypes QuickCheck `Arbitrary` instanties te maken. De hoofdreden hiervoor is dat Combles slechts efficiënt zijn als men veel waarden uit een enkele Comble haalt. Dit niet mogelijk met de `Arbitrary` typeclass. Een andere reden is dat QuickCheck de grootteparameter snel laat toenemen van een kleine waarde tot een zeer grote waarde. Het is echter onze bedoeling om voor elke grootte veel tests uit te voeren, zodat we de kleinste termen kunnen vinden waarvoor een bepaalde eigenschap faalt. Ook gebruiken grote Combles zeer veel geheugen.

5.1 System F

Voor System F hebben we twee eigenschappen getest:

***Progress*:** Een goedgetypeerde term is ofwel een waarde, ofwel kunnen we er een bèta-reductie op toepassen en is het resultaat ook goedgetypeerd.

***Preservation*:** De bèta-reductie van een reduceerbare goedgetypeerde term heeft hetzelfde type.

Onze uitbreiding genereert voor dit systeem 44 lijnen Haskell-code. Aangezien zeer kleine termen bijna altijd ofwel niet goed-getypeerd zijn ofwel niet verder reduceerbaar zijn, zijn de kleinste termen die we genereren van grootte 4. De grootste termen zijn van grootte 11, aangezien grotere Combles te veel geheugen gebruiken. Voor de grootte van de types is gekozen voor een afgerond logaritme van de termgrootte. Voor elke eigenschap en grootte werden 1000 termen die aan de precondities voldoen getest.

Gevonden fouten

De tests voor *progress* slaagden steeds. Bij *preservation* hebben we echter wel een fout gevonden die ervoor zorgde dat bepaalde termen na een evaluatiestap niet hetzelfde type hadden als ervoor. Een voorbeeld van een foute evaluatiestap is

$$(\Lambda\chi : \lambda x^{\chi} : \lambda y^{\chi} : x)\, B$$
$$\Rightarrow\ \lambda x^{B} : \lambda y^{\chi} : x$$

Het is duidelijk dat het resultaat $\lambda x^B : \lambda y^B : x$ moet zijn. De typevariabele in de binnenste abstractie wordt niet vervangen en wordt een vrije variabele.

Dit was niet eenvoudig om op te lossen, aangezien het om meerdere fouten bleek te gaan. Deze fouten zaten zowel in de implementatie van System F als in bestaande delen van *Autbound*. Met behulp van ons systeem zijn we erin geslaagd om deze fouten iteratief op te lossen. Het debuggen werd vergemakkelijkt door de mogelijkheid om met ons systeem eenvoudig tegenvoorbeelden te genereren.

Tijd

Figuren 2 en 3 bevatten een voorbeeld van respectievelijk de benodigde tijd en de fractie van termen die aan de preconditie voldoen in functie van de termgrootte van de generator. Deze grafieken zijn gemaakt met een versie van de System F-implementatie waarin alle gevonden bugs zijn opgelost.

Zoals te zien in Figuur 2 lijkt de tijd nodig om een eigenschap te testen superlineair te stijgen met de grootte van de termen. Dit is echter slechts deels te wijten aan de tijd die de Comble nodig heeft om kardinaliteiten te berekenen. Figuur 3 toont dat de fractie van termen die aan de preconditie voldoen tussen alle termen ook superlineair daalt met de grootte van de termen. Dit verklaart waarom de tests voor de progress-eigenschap minder lang duren dan die voor de preservation-eigenschap: progress heeft een minder strenge preconditie. De benodigde tijd is dus in grote mate te wijten aan de zeldzaamheid van "interessante" termen. Aangezien de specificatie geen informatie bevat over de typeringsregels of wat een waarde is, is dit moeilijk om op te lossen.

5.2 F_i^+ en F_{co}

Voor dit systeem hebben we *preservation* en *progress* getest in F_{co}, aangezien de implementatie geen evaluatie voorziet voor F_i^+-termen. We hebben echter ook een andere eigenschap van F_i^+ getest, namelijk *elaboration*. Deze eigenschap van F_i^+ betekent dat elke goedgetypeerde F_i^+-term een overeenkomende goedgetypeerde F_{co}-term heeft.

F_{co}-termen zijn complexer dan System F-termen: er zijn drie datatypes in plaats van twee, en er zijn meer constructoren. Daarom is het slechts haalbaar gebleken om termen tot grootte 7 te genereren en testen. Ook F_i^+ is comlexer dan System F, maar aangezien het geen derde datatype bevat is het wel nog haalbaar om termen tot grootte 11 te testen. Zoals bij System F nemen we de grootte van de andere datatypes als een afgerond logaritme van de termgrootte. Vanwege de hogere complexiteit wordt er ook meer code gegenereerd: 72 lijnen voor F_{co} en 62 lijnen voor F_i^+. De benodigde tijd en fractie verworpen termen verlopen analoog aan System F.

Gevonden fouten

Aangezien er bij System F een fout in de codegenerator gevonden werd, verwachten we ook een fout bij dit systeem. Deze hebben we ook gevonden, bij zowel *preservation* als *elaboration*. Bij *preservation* werd er echter pas een tegenvoorbeeld gevonden na een aantal uitvoeringen van het testprogramma, dus na meer dan 1000 tests. De precieze oorzaak is bij dit systeem nog niet vastgesteld en opgelost.

6 Conclusies

We hebben een uitbreiding aan de *Autbound*-codegenerator geschreven die enumeraties van programma's opstelt op basis van een specificatie. Deze specificaties hebben we gebruikt om eigenschappen van implementaties van programmeertalen te testen en bugs te vinden. Deze aanpak heeft ook geholpen bij het oplossen van deze bugs. We kunnen de onderzoekvraag dus positief beantwoorden.

We kunnen efficiënt vele programma's van dezelfde grootte genereren, maar deze groottes zijn beperkt. De gegenereerde enumeraties zijn ook te groot om exhaustief te testen, en dus zijn willekeurige bemonsteringen van de enumeratie nodig. Deze problemen hebben echter weinig invloed op de effectiviteit van onze aanpak.

Ook kan het moeilijk zijn om voldoende termen te vinden die aan de preconditie van een eigenschap voldoen. Zeker voor precondities die bij bijna alle eigenschappen voorkomen, zoals goedgetypeerdheid, is dit een probleem. Het systeem genereert een grote hoeveelheid termen die niet goedgetypeerd en dus niet interessant zijn. Dit is echter een limitatie van de gebruikte specificatietaal, en automatische generatie van enkel "interessante" termen zou een veel complexere specificatietaal en codegenerator vergen.

7 Toekomstig werk

7.1 Memoized Combles

Een Comble voor termen van grootte n zal vaak op meerdere plaatsen identieke Combles voor termen van grootte $n-1$ bevatten, bijvoorbeeld in de body van een typeapplicatie en in de body van een functieapplicate in System F.

In de huidige gegenereerde code worden deze subbomen telkens opnieuw gegenereerd en opgeslagen, wat zowel tijd als geheugen verspilt. Het zou zeer interessant zijn om memoized code te genereren die iedere unieke subboom slechts één keer genereert en opslaagt.

Dit is echter niet triviaal, aangezien een Comble naast potentieel veel verschillende grootteparameters ook een *environment* van gebonden variabelen als argument heeft.

7.2 Typed generator

De *Autbound*-specificatietaal ontbreekt veel informatie over hoe datatypes van elkaar afhangen, zoals typeringsregels. De meeste eigenschappen zijn echter enkel van toepassing op goedgetypeerde termen. Het feit dat we steeds veel termen genereren die niet aan deze basiseigenschappen voldoen beperkt dus de efficiëntie van onze tests.

Daarom zou een codegenerator om enumeraties op te stellen van enkel goedgetypeerde termen zeer interessant zijn. Nog interessanter is een functie die een enumeratie opstelt van alle termen van een bepaalde grootte die een bepaald type hebben.

Initiële experimenten met een handgeschreven generator van dit type tonen dat deze aanpak veelbelovend is. Hieruit blijkt echter ook dat deze generatoren complexer zijn en een uitgebreidere specificatietaal vereisen. Zulk een generator moet ook omgaan met het feit dat er niet voor elk type termen bestaan.

7.3 Backtracking generator

Een andere, mogelijks algemenere aanpak voor het genereren van "interessante" termen is om gebruik te maken van de boomstructuur van een Comble. We doorlopen de Comble vanaf de root voor elke term. Veel van deze termen voldoen echter niet aan de precondities. Mogelijks is het efficiënter om, in plaats van de Comble steeds vanaf de root te doorlopen, te backtracken en te proberen een term te vinden die wel aan de precondities voldoet.

7.4 Shrinking

Een feature van QuickCheck die niet geïmplementeerd is in onze codegenerator is "shrinking". Wanneer QuickCheck een tegenvoorbeeld vindt voor een eigenschap zal het de test opnieuw proberen met een kleinere input, gebaseerd op het tegenvoorbeeld. Dit vergemakkelijkt het debuggen, aangezien een tegenvoorbeeld zo minder irrelevante details zal hebben.

Referenties

[1] B. Belpaire. “Abstract Syntax Tree Code Generator for Haskell-based Com- and Transpilers”. Masterscriptie. Leuven: KU Leuven, 2019.

[2] X. Bi e.a. “Distributive Disjoint Polymorphism for Compositional Programming”. In: deel 11423. Springer Verlag, 2019, p. 381–409.

[3] A. Church en J. B. Rosser. “Some properties of conversion”. eng. In: *Transactions of the American Mathematical Society* 39.3 (1936), p. 472–482.

[4] K. Claessen en J. Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs”. eng. In: *ACM SIGPLAN Notices* 46.4 (2011), p. 53–64.

[5] J. Duregård, P. Jansson en M. Wang. “Feat: functional enumeration of algebraic types”. eng. In: *ACM SIGPLAN Notices* 47.12 (2013), p. 61–72.

[6] F. Kamareddine. “Reviewing the Classical and the de Bruijn Notation for calculus and Pure Type Systems”. In: *Journal of Logic and Computation* 11.3 (2001), p. 363–394.

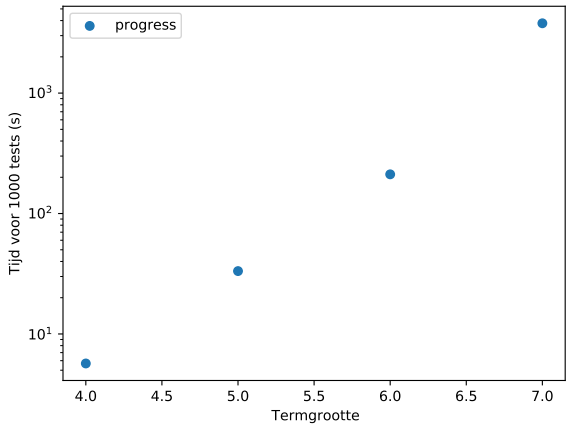
[7] C. McBride en R. Paterson. “Applicative programming with effects”. In: *Journal Of Functional Programming* 18 (2008), p. 1–13.

[8] John C. Reynolds. “Towards a theory of type structure”. In: *Programming Symposium*. Red. door B. Robinet. Springer, 1974, p. 408–425.

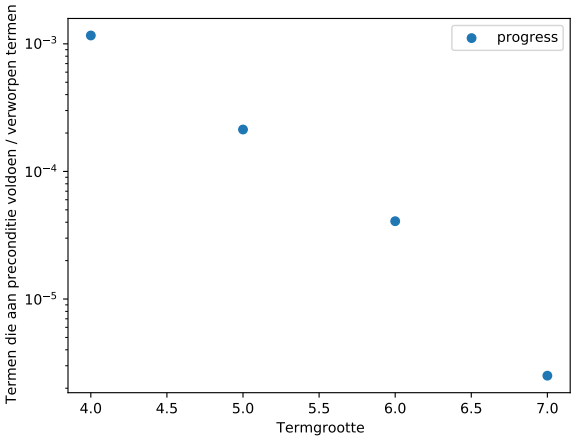
[9] C. Runciman, M. Naylor en F. Lindblad. “Smallcheck and lazy smallcheck: automatic exhaustive testing for small values”. eng. In: *ACM SIGPLAN Notices* 44.2 (2009), p. 37–48.

A Benodigde tijd voor F_i^+ en F_{co}

De fouten in deze implementaties zijn nog niet opgelost. Aangezien ons testprogramma slechts het kleinste tegenvoorbeeld zoekt voor elke eigenschap, betekent dit dat we enkel voor *progress* in F_{co} data hebben. Toch is het uit Figuren 4 en 5 duidelijk dat voor deze eigenschap zowel de tijd en fractie termen die aan de preconditie voldoen verlopen zoals bij System F.



Figuur 4: De tijd nodig om 1000 tests voor *progress* in F_{co} uit te voeren in functie van de grootte van de gegenereerde termen.



Figuur 5: De fractie van van termen in F_{co} die aan de preconditie van *progress* voldoen in functie van de grootte van de gegenereerde termen.