



Faculteit Toegepaste Wetenschappen

Vakgroep Informatietechnologie

Voorzitter: Prof. Dr. ir. P. LAGASSE

Systematisch ontwerp van XML-hulpmiddelen in een functionele taal

door

Hannes VERLINDE

Promotor: Prof. Dr. ir. R. BOUTE

Scriptie ingediend tot het behalen van de academische graad van
licentiaat in de informatica

Academiejaar 2002–2003



Faculteit Toegepaste Wetenschappen

Vakgroep Informatietechnologie

Voorzitter: Prof. Dr. ir. P. LAGASSE

Systematisch ontwerp van XML-hulpmiddelen in een functionele taal

door

Hannes VERLINDE

Promotor: Prof. Dr. ir. R. BOUTE

Scriptie ingediend tot het behalen van de academische graad van
licentiaat in de informatica

Academiejaar 2002–2003

Voorwoord

Graag wil ik mijn promotor, Prof. Boute, bedanken voor de wekelijkse feedbacksessies, die steeds opnieuw een bron waren van nieuwe inzichten en uitdagingen.

Toelating tot bruikleen

De auteur geeft de toelating deze scriptie voor consultatie beschikbaar te stellen en delen van de scriptie te kopiëren voor persoonlijk gebruik.

Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze scriptie.

Hannes Verlinde, mei 2003

Systematisch ontwerp van XML-hulpmiddelen in een functionele taal

door

Hannes VERLINDE

Promotor: Prof. Dr. ir. R. BOUTE

Scriptie ingediend tot het behalen van de academische graad van
licentiaat in de informatica

Academiejaar 2002–2003

Faculteit Toegepaste Wetenschappen
Universiteit Gent

Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. ir. P. LAGASSE

Samenvatting

We presenteren een systematische methode voor de ontwikkeling van programmatuur met behulp van functionele talen. Hierbij wordt het declaratieve formalisme Funmath gebruikt voor het opstellen van formele specificaties, gebaseerd op een informele beschrijving van de gewenste functionaliteit. Uit deze formele beschrijving kan dan een eerste implementatie in de functionele programmeertaal Haskell afgeleid worden. Haskell blijkt bijzonder geschikt voor de implementatie van XML-hulpmiddelen, onder meer omdat de beperkte typeringstaal van DTD's eenvoudig kan ingebed worden in het typemechanisme van Haskell.

Dit alles wordt geïllustreerd aan de hand van de ontwikkeling van een DTD-afhankelijke editor voor XML-documenten. Een dergelijke editor garandeert dat de gewijzigde documenten geldig zijn met betrekking tot een gegeven DTD door enkel die wijzigingen toe te laten die de correctheid niet kunnen aantasten. De ontwikkelde editor is een structurele XML-editor. Dit houdt in dat als basiselementen niet de individuele tekens van een tekstbestand beschouwd worden, maar wel de syntactische basisconstructies van een XML-document. Ook het grafische aspect van de editor en de interactie met de gebruiker worden behandeld. We illustreren hoe een grafische gebruikersinterface op declaratieve wijze kan geïmplementeerd worden in Haskell.

Trefwoorden: XML, Haskell, Funmath

Inhoudsopgave

1	Inleiding	1
1.1	XML	1
1.1.1	Documentstructuur en syntax	2
1.1.2	DTD's en grammaticale correctheid	3
1.1.3	Verwante technologieën	5
1.2	Funmath	5
1.2.1	Conventies	6
1.3	Haskell	9
2	Een DTD-afhankelijke XML-editor	12
2.1	Doel en gebruik van een structurele editor	12
2.2	Formele specificaties	13
2.2.1	Interne voorstelling	14
2.2.2	Formele beschrijving van DtdToHaskell	15
2.2.3	Formele beschrijving van de editor-functionaliteit	23
3	Implementatie in Haskell	38
3.1	Implementatie van de editor-functionaliteit	38
3.2	Implementatie van de grafische gebruikersinterface	44
3.2.1	Basisconcepten	45
3.2.2	De listener-operatoren	47
3.2.3	De event-operatoren	48
3.2.4	De editor GUI	50
4	Gerelateerd onderzoek	59
4.1	XML en Haskell	59

4.1.1	HaXml	59
4.1.2	Haskell XML Toolbox	60
4.2	XSLT en Haskell	60
4.3	Web scripting in Haskell	61
4.4	Generic Haskell	61
4.5	XML en Funmath	62
5	Conclusies	63
	Referenties	67

Hoofdstuk 1

Inleiding

Het hoofddoel van deze scriptie is tweeledig. Enerzijds presenteren we een systematische ontwerpmethodologie voor de ontwikkeling van programmatuur, waarbij gebruik gemaakt wordt van het declaratieve formalisme Funmath. Anderzijds illustreren we dat een moderne functionele programmeertaal, in casu Haskell, uitermate geschikt is voor de verwerking van XML en verwante technologieën. In hoofdstuk 1 wordt een beknopt overzicht gegeven van XML, Funmath en Haskell. We bespreken ook de gelijkenissen en de verschillen tussen deze talen. In hoofdstuk 2 beschrijven we de ontwikkeling van een grafische editor voor XML-documenten. Vertrekkende van een informele beschrijving van de functionaliteit ontwerpen we vooreerst specificaties in Funmath. Vervolgens baseren we ons in hoofdstuk 3 op deze formele specificaties voor het uitwerken van een concrete implementatie in Haskell. Extra aandacht wordt besteed aan het declaratieve ontwerp van de grafische gebruikersinterface. In hoofdstuk 4 bespreken we gerelateerd onderzoek en suggereren we mogelijke uitbreidingen voor het geïmplementeerde prototype. In een laatste hoofdstuk geven we een overzicht van de belangrijkste resultaten en formuleren we de conclusies.

1.1 XML

Een opmaaktaal laat toe structuur toe te voegen aan documenten door middel van opmaakcodes. Voorbeelden van opmaaktalen zijn de **Standard Generalized Markup Language** (SGML), de **Hypertext Markup Language** (HTML) en uiteraard ook de **Extensible Markup Language** (XML). SGML is een zeer complexe en omvangrijke opmaaktaal die moeilijk aan te leren en te implementeren valt. HTML is een concrete toepassing van SGML en wordt

vooral gebruikt voor de opmaak van webpagina's. Door de vaste verzameling opmaakcodes is het toepassingsgebied van HTML eerder beperkt. XML is een restrictie van SGML die vrij eenvoudig kan geïmplementeerd worden maar toch zeer flexibel is. Door het gebruik van XML kan men tekstdocumenten voorzien van annotaties onder de vorm van opmaakcodes, die expliciet de structuur van de data beschrijven. Bovendien is XML uitbreidbaar, wat inhoudt dat de auteur van een XML-document zelf de gebruikte opmaakcodes vrij kan kiezen. De opmaakcodes kunnen dus zo gekozen worden dat ze de inhoud van het document verduidelijken voor wie het document inkijkt. De XML syntax blijft echter eenvoudig en vast bepaald, zodat XML-documenten ook door een computer gemakkelijk kunnen verwerkt worden. Deze combinatie van flexibiliteit en uniformiteit verklaart wellicht het grote succes van XML-gerelateerde technologieën. XML wordt onder meer gebruikt bij informatie-overdracht op het Internet, voor platformonafhankelijke communicatie tussen applicaties en voor gestructureerde informatie in databanken. In 1998 werd XML gestandaardiseerd door het **World Wide Web Consortium** (W3C) [7].

Een grondige inleiding tot XML en verwante technologieën is te vinden in [21, 27, 28]. Voor meer gedetailleerde referentiewerken verwijzen we naar [17, 39].

1.1.1 Documentstructuur en syntax

De XML syntax steunt op een eenvoudig, hiërarchisch datamodel, waarbij elk document kan worden voorgesteld door een boomstructuur. Een document dat voldoet aan alle XML syntaxregels noemen we een **goed gevormd** of **syntactisch correct** XML-document. Een goed gevormd XML-document bevat naast een optionele proloog en epiloog een elementenboom met juist één wortel, het **documentelement**. De elementen van deze boom corresponderen met de logische, structurele basiselementen van het document. De data van elk element bestaat uit nieuwe elementen of karakterdata, voorafgegaan door een start-tag en gevolgd door een eind-tag:

```
<element> data </element>
```

Een leeg element is een element zonder data. Hiervoor bestaat een verkorte notatie:

```
<element />
```

Aan de verschillende elementen kunnen attributen toegevoegd worden:

```
<element attribuut="waarde"> data </element>
```

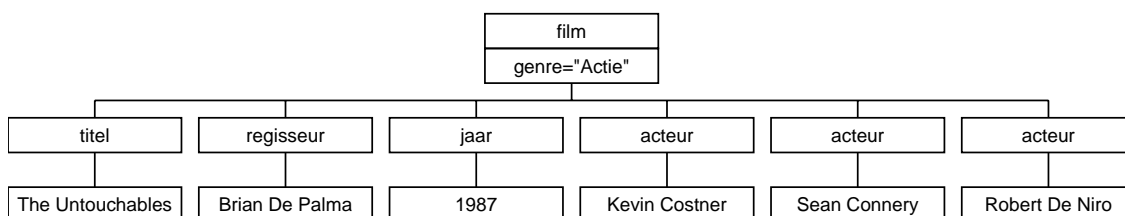
Wanneer men informatie aan een element wil toevoegen, kan men opteren voor een attribuut of voor een nieuw, onderliggend element. Deze keuze is in zekere mate arbitrair. Een vuistregel

stelt dat attributen kunnen gebruikt worden voor het toevoegen van meta-informatie maar dit is geen bindende regel [28]. Een meer gedetailleerde beschrijving van de XML syntaxregels is te vinden in [7]. Fig. 1.1 geeft een voorbeeld van een goed gevormd XML-document. Merk op dat de

```
<?xml version="1.0"?>
<film genre="Actie">
<titel>The Untouchables</titel>
<regisseur>Brian De Palma</regisseur>
<jaar>1987</jaar>
<acteur>Kevin Costner</acteur>
<acteur>Sean Connery</acteur>
<acteur>Robert De Niro</acteur>
</film>
```

Figuur 1.1: Een goed gevormd XML-document

opmaakcodes zo gekozen zijn dat het XML-document zelfverklarend wordt. De overeenkomstige elementenboom is afgebeeld in Fig. 1.2.



Figuur 1.2: De elementenboom

1.1.2 DTD's en grammaticale correctheid

Een **Document Type Definition** (DTD) [7] definieert de structuur van een bepaalde klasse XML-documenten. Zo kan een gebruiker zelf de opmaakcodes, de bijhorende attributen en de onderlinge volgorde tussen de elementen vastleggen. Een voorbeeld van een syntactisch correcte DTD is gegeven in Fig. 1.3. Met elk soort element uit de elementenboom correspondeert in de DTD een **!ELEMENT**-tag. Indien een element als data geen nieuwe elementen kan bevatten, spreken we van een **terminaal element**. Een dergelijk element bevat dan enkel karakterdata of

is verplicht leeg. In Fig. 1.3 worden `titel`, `regisseur`, `jaar` en `acteur` als terminale elementen gedefinieerd. Voor een **niet-terminaal element** kunnen de onderliggende elementen — de mogelijke **opvolgers** in de elementenboom — een opeenvolging of een lijst van alternatieven vormen. Dit wordt in de DTD aangeduid door deze elementen te scheiden door respectievelijk een komma of een verticale streep. Ook gecombineerde, recursieve constructies kunnen gespecificeerd worden. Een optioneel element wordt aangeduid door een vraagteken en een mogelijke herhaling van elementen wordt genoteerd door een ster- of een plus-symbool, naar analogie met BNF syntax. Het element `film` is in Fig. 1.3 het enige niet-terminale element. De verschillende attributen voor een bepaald element kunnen gedefinieerd worden met behulp van een enkele of meerdere `!ATTLIST`-tags. De volgorde van deze attributen is irrelevant en kan dan ook niet vastgelegd worden in de DTD. Attributen kunnen verplicht (`#REQUIRED`), optioneel (`#IMPLIED`) of vast (`#FIXED`) zijn. Bij elk voorkomen van een element moeten steeds alle bijhorende verplichte attributen ingevuld worden. De optionele attributen mogen eventueel weggelaten worden. De vaste attributen hebben steeds dezelfde waarde, die bepaald wordt in de DTD. De waarde van een attribuut kan geen nieuwe elementen of attributen bevatten en bestaat over het algemeen uit karakterdata. Men kan echter ook, zoals voor het attribuut `genre` in Fig. 1.3, een enumeratietype definiëren.

```
<!ELEMENT film (titel, regisseur, jaar?, acteur*)>
<!ATTLIST film genre (Actie | Drama | Komedie | Thriller) #REQUIRED>
<!ELEMENT titel (#PCDATA)>
<!ELEMENT regisseur (#PCDATA)>
<!ELEMENT jaar (#PCDATA)>
<!ELEMENT acteur (#PCDATA)>
```

Figuur 1.3: Een DTD voor films

Zodra we beschikken over een DTD voor een bepaalde klasse XML-documenten, kunnen we voor goed gevormde documenten na de syntactische controle ook een **grammaticaal nazicht** uitvoeren. We zeggen dat een goed gevormd XML-document **grammaticaal correct** of kortweg **geldig** is met betrekking tot een gegeven DTD als de elementenboom van het document voldoet aan de specificaties uit de DTD. Het gebruik van DTD's voor een grammaticale controle van XML-documenten laat toe naast syntactische fouten ook specifieke, toepassingsafhankelijke fouten te ontdekken, doordat voor elke toepassing een afzonderlijke DTD kan gebruikt worden.

1.1.3 Verwante technologieën

Bij de ontwikkeling van XML zijn verschillende uitbreidingen aan de taal toegevoegd en is er een hele familie verwante technologieën ontstaan. **XML Namespaces** [6] is een mechanisme om naamconflicten te vermijden. De **XML Linking Language** (XLink) [13] en de **XML Pointer Language** (XPointer) [12] kunnen respectievelijk gebruikt worden om documenten te koppelen en bepaalde delen van een XML-document te adresseren. De **Extensible Stylesheet Language** (XSL) [1] wordt gebruikt om transformaties uit te voeren op XML-documenten en om de visuele opmaak vast te leggen. Zowel XPointer als XSL gebruiken de **XML Path Language** (XPath) [8] om locaties in een XML-document te specificeren. Ten slotte vermelden we nog **XML Schema** [15], een technologie die ontworpen werd om de tekortkomingen van DTD's op te vangen. Een XML-schema kan, net zoals een DTD, gebruikt worden om de structuur van een klasse XML-documenten te specificeren. De syntax van een dergelijk schema is echter goed gevormde XML, terwijl DTD's gebruik maken van een aparte syntax. Bovendien is de uitdrukkingskracht van XML Schema groter dan die van DTD's. Omwille van technische redenen zullen we bij het ontwerp van een XML-editor gebruik maken van DTD's. De aangebrachte ideeën zijn echter algemeen toepasbaar en steunen vooral op het algemene principe van grammaticale correctheid.

1.2 Funmath

Funmath is een wiskundig formalisme dat onder meer kan gebruikt worden om functionaliteit van programmatuur te modelleren en er formeel over te redeneren. Door de declaratieve aard van het formalisme kan dit op een abstract niveau gebeuren, losstaand van een concrete implementatie in een bepaalde programmeertaal. Zo gebruiken we Funmath bij de ontwikkeling van een XML-editor als een brug tussen de informele beschrijving van de functionaliteit en een concrete implementatie in Haskell. Deze aanpak dwingt ons om in eerste instantie heel precies en formeel de gewenste functionaliteit te karakteriseren, zonder ons reeds te bekommeren om implementatiedetails. Het formalisme is flexibel en universeel, zodat deze omzetting vrij intuïtief kan gebeuren. De resulterende specificatie van de functionaliteit in Funmath vormt een algemene referentie voor een mogelijke implementatie, en kan bovendien gebruikt worden om bepaalde eigenschappen formeel aan te tonen. Indien bijvoorbeeld vooraf bepaalde eisen gesteld worden aan de programmatuur, dan kan men in deze fase formeel verifiëren of aan de eisen voldaan is.

Op die manier kan men ontwerpfouten in een vroeg stadium van de ontwikkeling elimineren en de extra implementatiekosten vermijden.

1.2.1 Conventies

Het Funmath formalisme vormt een concrete syntax voor het meer algemene **Functional mathematics** principe, waarbij wiskundige objecten waar mogelijk als functies gedefinieerd worden. Door de orthogonale combinatie van slechts vier syntactische constructies — identifier, functie-applicatie, getypeerde functie-abstractie en tupeldenotatie — worden op die manier de gebruikelijke wiskundige notaties gesynthetiseerd. Hierdoor kunnen de dubbelzinnigheden en inconsistenties in de traditionele notaties geëlimineerd worden en kunnen de formele rekenregels toegepast worden in een zeer algemene context [3].

Functies als basisconcept

Een functie f is volledig bepaald door haar domein $\mathcal{D}f$ en een beeldpuntdefinitie die met elke x uit $\mathcal{D}f$ een waarde fx associeert. Dit houdt onder meer in dat twee functies f en g aan elkaar gelijk zijn dan en slechts dan als $\mathcal{D}f = \mathcal{D}g$ en $\forall x : \mathcal{D}f \cap \mathcal{D}g . fx = gx$.

Een functie kan in sommige gevallen expliciet voorgesteld worden door een abstractie van de vorm $x : X . e$. Het domein van een dergelijke functie is de verzameling X en elke x uit X wordt afgebeeld op de uitdrukking e , die afhankelijk kan zijn van de variabele x . Indien x niet vrij voorkomt in e schrijven we $X \bullet e$ voor $x : X . e$. Een verfijnde vorm van een abstractie is $x : X \wedge p . e$, met p een propositie die gebruikt wordt als filter voor het domein: $x \in \mathcal{D}(x : X \wedge p . e) \equiv x \in X \wedge p$.

Het bereik van een functie wordt gegeven door de operator \mathcal{R} , die gedefinieerd is door

$$y \in \mathcal{R}f \equiv \exists x : \mathcal{D}f . y = fx.$$

Indien we de alternatieve notatie $\{f\}$ voor $\mathcal{R}f$ combineren met functie-abstracties, krijgen we een vertrouwde notatie voor verzamelingen. Zo stelt de uitdrukking $\{n : \mathbb{N} . n^2\}$ de verzameling van alle gehele kwadraten voor. De gebruikelijke ZF-schrijfwijzen voor verzamelingen kunnen we formaliseren door de volgende alternatieve notaties voor abstracties in te voeren:

$$e \mid x : X \text{ voor } x : X . e \text{ en } x : X \mid p \text{ voor } x : X \wedge p . x.$$

Op die manier worden vertrouwde uitdrukkingen zoals $\{n:\mathbb{N} \mid n < m\}$ en $\{2 \cdot n \mid n:\mathbb{N}\}$ ontdaan van alle potentiële dubbelzinnigheid en kunnen via de definitie voor functiebereik de formele rekenregels van de functionele predikaten toegepast worden.

We noteren $A \rightarrow B$ voor de verzameling functies met domein A en bereik behorende tot B :

$$f \in A \rightarrow B \equiv \mathcal{D}f = A \wedge \mathcal{R}f \subseteq B.$$

Een predikaat is een functie met bereik $\mathbb{B} = \{0, 1\}$. De kwantoren \forall en \exists zijn zelf predikaten over predikaten, en zijn voor een willekeurig predikaat P gedefinieerd door

$$\forall P \equiv P = \mathcal{D}P \bullet 1 \quad \text{en} \quad \exists P \equiv P \neq \mathcal{D}P \bullet 0.$$

Uit deze definities volgt een uitgebreid pakket algebraïsche rekenregels, dat hier echter niet verder besproken wordt. De combinatie van kwantoren met predikaten onder de vorm van abstracties levert opnieuw vertrouwde uitdrukkingen zoals $\forall n:\mathbb{N}. n \geq 0$.

Tupels, lijsten en rijen worden gedefinieerd als functies met domein $\square n = \{m:\mathbb{N} \mid m < n\}$ met n in $\mathbb{N} \cup \infty$. Dergelijke functies kunnen voorgesteld worden met behulp van tupeldenotatie. Zo heeft de functie a, b, c als domein $\square 3$ en is de beeldpuntdefinitie bepaald door

$$(a, b, c) 0 = a \quad (a, b, c) 1 = b \quad (a, b, c) 2 = c.$$

Merk op dat $\forall(a, b) = a \wedge b$ voor willekeurige a en b in \mathbb{B} . De singleton-tupel-operator τ wordt gedefinieerd door $\tau x = 0 \mapsto x$ voor willekeurige x . De lengte $\#t$ van een tupel (lijst, rij) t wordt gedefinieerd door $\#t = n \equiv \mathcal{D}t = \square n$. De verzameling van alle lijsten of rijen met lengte n waarvan alle elementen tot de verzameling A behoren is de functieverzameling $\square n \rightarrow A$, alternatief genoteerd door A^n . Een tweedimensionale rij of matrix met m rijen en n kolommen noteren we verkort als $A^{m,n} = (A^n)^m = \square m \rightarrow \square n \rightarrow A$. De verzameling van alle eindige lijsten of rijen over A wordt gegeven door $A^* = \bigcup n:\mathbb{N}. A^n$. Verder geldt ook $A^\infty = \mathbb{N} \rightarrow A$ en definiëren we $A^\omega = A^* \cup A^\infty$.

Generische functionalen

Binnen het Funmath formalisme beschikken we over een collectie functionalen die zorgvuldig ontworpen zijn met het oog op algemeen gebruik. Omdat wiskundige objecten in Funmath uniform als functies worden gedefinieerd, is het toepassingsgebied van dergelijke functionalen

bijzonder groot. We bespreken nu enkele generische functionalen die in hoofdstuk 2 frequent zullen gebruikt worden.

Naast de reeds behandelde constante-functie-functionaal \bullet zijn er twee andere triviale basisfunctionalen ε en \mapsto . De lege functie wordt gedefinieerd door $\varepsilon = \emptyset \bullet e$. Merk op dat $\emptyset \bullet e = \emptyset \bullet e'$, ongeacht e en e' . De één-puntsfunctie-functionaal wordt voor willekeurige x en y gedefinieerd door $x \mapsto y = \iota x \bullet y$, waarbij ι de singleton-functie voorstelt met axioma $z \in \iota x \equiv z = x$.

Voor een willekeurige functie f (niet noodzakelijk injectief) wordt de inverse gegeven door f^- met als axioma

$$\mathcal{D} f^- = \text{Bran } f \wedge \forall x : \text{Bdom } f . f^-(f x) = x$$

Het bijectieve domein Bdom en het bijectieve bereik Bran worden als volgt gedefinieerd:

$$\begin{aligned} \text{Bdom } f &= \{x : \mathcal{D} f \mid \forall x' : \mathcal{D} f . f x' = f x \Rightarrow x' = x\} \\ \text{Bran } f &= \{f x \mid x : \text{Bdom } f\} \end{aligned}$$

De samenstelling van twee functies f en g wordt met behulp van een functie-abstractie gedefinieerd door $f \circ g = x : \mathcal{D} g \wedge g x \in \mathcal{D} f . f (g x)$. Functietranspositie definiëren we als $f^\top = y : \bigcap (\mathcal{D} \circ f) . x : \mathcal{D} f . f x y$. Merk op dat voor tupels de eigenschappen

$$f \circ (x, y, z) = f x, f y, f z \quad \text{en} \quad (f, g, h)^\top x = f x, g x, h x$$

gelden mits $\{x, y, z\} \subseteq \mathcal{D} f$ en $x \in \mathcal{D} f \cap \mathcal{D} g \cap \mathcal{D} h$. Vaak worden functiesamenstelling en functietranspositie voor tupels gecombineerd tot uitdrukkingen van de vorm $f \circ s^\top$, wat we ook zullen noteren als $f^{<s}$. Er geldt dan bijvoorbeeld voor een functie f die een tuple als argument neemt en voor tupels a, b en c van lengte n dat $f^{<(a, b, c)} i = f (a i, b i, c i)$, met $i : \square n$.

De directe uitbreiding van een twee-argument-operator \star wordt voor willekeurige functies f en g gedefinieerd door $f \hat{\star} g = x : \mathcal{D} f \cap \mathcal{D} g \wedge (f x, g x) \in \mathcal{D} (\star) . (f x) \star (g x)$. Merk op dat $f \hat{\star} g = (\star) \circ (f, g)^\top$. In vele gevallen is een halve directe uitbreiding nodig, met als definitie $f \overleftarrow{\star} x = f \hat{\star} \mathcal{D} f \bullet x$ en $x \overrightarrow{\star} g = \mathcal{D} g \bullet x \hat{\star} g$.

Wanneer een functie g “voorrang” krijgt op een functie f kunnen we dit uitdrukken met behulp van de operator \otimes gedefinieerd door $f \otimes g = x : \mathcal{D} f \cup \mathcal{D} g . x \in \mathcal{D} f ? f x \dagger g x$. Hierbij wordt gebruik gemaakt van een conditionele expressie die voor c in \mathbb{B} en willekeurige uitdrukkingen a en b gedefinieerd wordt door $c ? b \dagger a = (a, b) c$.

De restrictie $f \upharpoonright X$ van een functie f tot een verzameling X is gedefinieerd door $f \upharpoonright X = x : \mathcal{D} f \cap X . f x$.

Twee functies f en g kunnen “versmolten” worden door de functionaal \cup , gedefinieerd door $f \cup g = x : \mathcal{D} f \cup \mathcal{D} g \wedge (x \in \mathcal{D} f \cap \mathcal{D} g \Rightarrow f x = g x) . (f \otimes g) x$. Vaak zullen we deze operator combineren met de één-puntsfunctie-functionaal \mapsto , die in dat geval een hogere prioriteit krijgt. De uitdrukking $a \mapsto b \cup c \mapsto d$ staat dus voor $(a \mapsto b) \cup (c \mapsto d)$.

Het functioneel Cartesisch product wordt gebruikt om een fijne typering voor functies te specificeren. Deze functionaal neemt als argument een familie verzamelingen $S \multimap S$ — S is dus een functie met $S x$ een verzameling voor elke x in $\mathcal{D} S$ — en wordt als volgt gedefinieerd:

$$\times S = \{f : \mathcal{D} S \rightarrow \bigcup S \mid \forall x : \mathcal{D} f \cap \mathcal{D} S . f x \in S x\}$$

Als bijzondere gevallen vermelden we $\times(A \bullet B) = A \rightarrow B$ en $\times(A, B) = A \times B$, waarbij $A \times B$ het klassieke Cartesisch product voorstelt, gedefinieerd door $(a, b) \in A \times B \equiv a \in A \wedge b \in B$. Een interessante toepassing van deze operator is de voorstelling van een afhankelijk type als $\times(a : A . B_a) = \{f : A \rightarrow \bigcup a : A . B_a \mid \forall a : A . f a \in B_a\}$, wat we over het algemeen zullen noteren als $A \ni a \rightarrow B_a$. Een andere toepassing zijn recordtypes, waarbij records gezien worden als functies over de enumeratieverzameling van de veldnamen, bijvoorbeeld $\times(\text{personeelslid} \mapsto \text{string} \cup \text{geboortjaar} \mapsto 1900..2099)$. Daarom zullen we uitdrukkingen van de vorm $\times(a \mapsto b \cup c \mapsto d)$ soms ook noteren als **Record** $(a \mapsto b, c \mapsto d)$.

Als laatste generische functionaal vermelden we de keuze-operator $|$ die kan gebruikt worden om een onderscheiden vereniging van een familie S van types te bepalen:

$$|S = \bigcup x : \mathcal{D} S . \{x \mapsto y \mid y : S x\}.$$

Dit naar analogie met $\bigcup S = \bigcup x : \mathcal{D} S . \{y \mid y : S x\}$ voor de “gewone” vereniging. Door $|$ wordt elke waarde van een type uit de vereniging voorzien van een label dat de elementen in S onderscheidt op grond van de verzameling waar zij uit afkomstig zijn. Uitdrukkingen van de vorm $|(a \mapsto b \cup c \mapsto d)$ zullen we ook noteren als **Choice** $(a \mapsto b, c \mapsto d)$. Merk op dat **Choice** $(a \mapsto b) = \text{Record}(a \mapsto b)$, wat later de eenvormigheid in de semantische definities zal bevorderen.

Een uitgebreide inleiding tot Funmath en de achterliggende principes is te vinden in [4, 5].

1.3 Haskell

Haskell is een zuiver functionele programmeertaal. In een functionele programmeertaal bestaat een programma uit een enkele uitdrukking, die geëvalueerd wordt bij de uitvoering. De specifieke

voordelen van Haskell in vergelijking met de meeste andere programmeertalen kunnen als volgt samengevat worden [24]:

- Een hogere productiviteit en kortere ontwikkelingsperiode
- Kortere, duidelijkere en gemakkelijker te onderhouden code
- Minder programmeerfouten en hogere betrouwbaarheid
- Een kleinere semantische barrière tussen de programmeur en de taal

De meest frequent genoemde, meer algemene voordelen van functionele programmeertalen ten opzichte van imperatieve programmeertalen zijn de beheersbaarheid van neveneffecten en de aanzienlijk eenvoudigere methodiek voor het bewijzen van eigenschappen. Er moet echter een onderscheid gemaakt worden tussen deze begrippen in de context van een functionele programmeertaal enerzijds en in de context van het eerder genoemde formalisme Funmath anderzijds. Bij functionele programmeertalen ligt de klemtoon meer dan bij traditionele programmeertalen op **wat** het programma als taak uitvoert en minder op **hoe** die taak precies wordt uitgevoerd, zodat functionele programmeertalen reeds een eerste stap zijn in de richting van declarativiteit. Het formalisme Funmath is echter zuiver declaratief en biedt een abstract redeneringsniveau dat volledig los staat van enige implementatie. Door de referentiële transparantie en het ontbreken van neveneffecten is een bewijs van correctheid veel eenvoudiger voor functionele dan voor imperatieve programma's. Funmath biedt daarenboven een heel gamma aan formele bewijstechnieken, wat het bewijzen van eigenschappen in een universeel wiskundig kader mogelijk maakt.

Zoals reeds gezegd biedt Haskell als programmeertaal een aantal exclusieve voordelen bij de ontwikkeling van programmatuur. Bovendien is onlangs gebleken dat Haskell zoals te verwachten uitermate geschikt is voor de verwerking van XML [38]. Een XML-document kan namelijk voorgesteld worden als een boomstructuur, en moderne functionele programmeertalen zoals Haskell zijn zeer sterk in het werken met recursieve datatypes en hiërarchisch gestructureerde data in het algemeen. Er is ook een sterke gelijkenis tussen de eenvoudige taal van DTD's en de rijkere typeringstaal van Haskell. Zo kan men een element-declaratie in een DTD interpreteren als de declaratie van een nieuw datatype. Een opeenvolging van onderliggende elementen kan men dan in Haskell voorstellen als een product-type en een lijst van alternatieven als een som-type. Een optioneel element wordt voorgesteld door een *Maybe*-type en de herhaling van elementen door

een Haskell-lijst. Op deze manier kan men een oppervlakkige inbedding (shallow embedding) van (getypeerde) XML in Haskell realiseren, wat een zeer elegante verwerking met zich meebrengt.

Het gebruik van een functionele programmeertaal zoals Haskell biedt ook extra voordelen in combinatie met het formalisme Funmath. Hoewel een specificatie in Funmath in principe volledig losstaat van een concrete implementatie, blijkt de omzetting naar een implementatie in Haskell veel eenvoudiger dan het geval zou zijn bij een imperatieve programmeertaal. De verklaring hiervoor is dat zowel Funmath als Haskell het functiebegrip als basisconcept gebruiken en dat heel wat executeerbare Funmath-constructies een corresponderende Haskell-representatie hebben. Haskell staat dan ook bekend als een **uitvoerbare specificatietaal**, en de stap tussen een zuiver declaratieve specificatie in Funmath en een uitvoerbare specificatie in Haskell is vaak verrassend klein.

Een inleiding tot Haskell is te vinden in [2], [20] en [22]. De technische specificatie van de taal staat beschreven in [25]. Uitgebreide documentatie en implementaties van de taal zijn te vinden via <http://www.haskell.org/>.

Hoofdstuk 2

Een DTD-afhankelijke XML-editor

2.1 Doel en gebruik van een structurele editor

Een XML-document is een tekstbestand waarbij een scheiding wordt gemaakt tussen de eigenlijke inhoud enerzijds en de structuur onder de vorm van opmaakcodes anderzijds. XML-bestanden kunnen dus bewerkt worden met een gewone teksteditor, maar deze aanpak wordt door velen als omslachtig ervaren. Reden hiervoor is dat het manueel wijzigen van XML-documenten op tekstniveau technische kennis van XML vereist, veel repetitief werk met zich meebrengt en bijzonder gevoelig is voor tikfouten. Om te verzekeren dat een XML-document syntactisch en grammaticaal correct is, kan na elke wijziging een geautomatiseerde controle uitgevoerd worden. Een gespecialiseerde XML-editor daarentegen laat toe XML-documenten op een meer intuïtieve manier te wijzigen, en maakt het valideren van gewijzigde XML-bestanden overbodig door geen manipulaties toe te laten die de correctheid kunnen aantasten. Om deze doelstellingen te bereiken, ontwerpen we een **structurele** editor, waarbij als basiselementen niet de individuele tekens van een tekstbestand beschouwd worden, maar wel de syntactische basisconstructies van een XML-document.

De kennis van de XML syntax kan eenmalig worden ingebouwd in de editor, maar voor het opleggen van de grammaticale correctheid zal de editor gebruik moeten maken van een extern gedefinieerde specificatie, onder de vorm van een DTD of een schema. De basisfunctionaliteit van een dergelijke DTD-afhankelijke editor kunnen we informeel als volgt beschrijven.

Vooreerst zijn er functies voor het openen en bewaren van documenten. Het openen van een bestaand document lukt enkel indien het XML-bestand syntactisch en grammaticaal correct is.

Bij het aanmaken van een nieuw document, wordt geen leeg document aangemaakt, maar wel een **minimaal** document dat voldoet aan de DTD-specificaties.

Verder is er ook een functie voor het weergeven van de inhoud van een document. Deze functie genereert naast pure tekst ook extra informatie, die bijvoorbeeld kan gebruikt worden voor het opmaken van de tekst of om te navigeren in het document.

De kern van de editor-functionaliteit bestaat uit het wijzigen van een document. Dit gebeurt in twee fasen. In een eerste fase selecteert de gebruiker een locatie in het document en worden de mogelijke wijzigingen aan de gebruiker gepresenteerd. Vervolgens kiest de gebruiker een specifieke wijziging die tenslotte wordt doorgevoerd. Het aantal mogelijke wijzigingen dat geen betrekking heeft op karakterdata is, voor een gegeven locatie in het document, over het algemeen voldoende beperkt om de gewenste wijziging door de gebruiker uit een menu te laten kiezen. Voor het wijzigen van karakterdata in een XML-document kunnen we een eenvoudige karaktergeoriënteerde teksteditor inbouwen in de structurele editor.

2.2 Formele specificaties

Een editor voor een datatype kunnen we kenschetsen als programmatuur die toelaat waarden van dat datatype te manipuleren. Het datatype dat we wensen te manipuleren is de klasse van goed gevormde XML-documenten die voldoen aan een gegeven grammaticale specificatie. Indien we als specificatie een DTD gebruiken, krijgen we zo een DTD-afhankelijke editor. De basisfunctionaliteit hiervan zullen we in eerste instantie modelleren in Funmath.

Als lopend voorbeeld zullen we de eenvoudige DTD uit hoofdstuk 1 (hier geïnterpreteerd als een tekstbestand van het type \mathbb{A}^*) gebruiken:

```
def filmdtd := “<!ELEMENT film (titel, regisseur, jaar?, acteur*)>  
    <!ATTLIST film genre (Actie|Drama|Komedie|Thriller) #REQUIRED>  
    <!ELEMENT titel (#PCDATA)>  
    <!ELEMENT regisseur (#PCDATA)>  
    <!ELEMENT jaar (#PCDATA)>  
    <!ELEMENT acteur (#PCDATA)>”
```

(2.1)

Omdat we bij de uiteindelijke implementatie gebruik zullen maken van de HaXml-toolkit, zullen we vooraf de interne werking van deze toolkit beschrijven, zodat we onze eigen specificaties hierop kunnen afstemmen. Zo volgen we bij de naamgeving van functies en labels vrijwel steeds de HaXml conventies. Voor onze doeleinden is vooral het programma `DtdToHaskell` uit de toolkit van belang. Dit programma neemt als input een DTD en genereert op basis hiervan een Haskell datatype. Dit datatype stelt de klasse XML-documenten voor die grammaticaal correct zijn met betrekking tot de gegeven DTD. Verder worden ook de nodige functies gegenereerd voor het inlezen en uitschrijven van waarden van dit datatype.

2.2.1 Interne voorstelling

Intern wordt een DTD na het parseren omgezet in een rij typedefinities, corresponderend met de specificaties in het oorspronkelijke DTD-bestand. De typedefinities zelf hebben het type `Typedef`, dat we als volgt definiëren:

$$\begin{aligned} \text{def Typedef} := \text{Record} \left(\begin{array}{l} \textit{name} \mapsto \mathbb{A}^+, \\ \textit{attributes} \mapsto \text{StructType}^*, \\ \textit{constructors} \mapsto (\text{StructType}^*)^+ \end{array} \right), \end{aligned} \quad (2.2)$$

Elke typedefinitie stelt de structuur van één soort XML-element voor, waarbij de labels *name*, *attributes* en *constructors* respectievelijk corresponderen met de naam, de attributen en de onderdelen van het element. Omdat deze labels frequent gebruikt zullen worden, gebruiken we voortaan de respectievelijke afkortingen *n*, *a* en *c*. We nemen als conventie aan dat voor een gegeven rij typedefinities, de eerste typedefinitie de structuur van het documentelement — de wortel van de elementenboom — beschrijft. Op die manier kunnen we de typedefinitie van dit speciale element steeds eenvoudig terugvinden. Zoals besproken in 1.1.2 kunnen de onderliggende elementen van een niet-terminaal element een opeenvolging of een lijst van alternatieven vormen. We herkennen hierin de analogie met product- en somtypes in Haskell. Daarom wordt het label *constructors* afgebeeld op een tweedimensionale rij. De elementen van deze rij stellen de verschillende alternatieven voor. Elk van deze alternatieven vormt opnieuw een rij die een opeenvolging van onderliggende elementen voorstelt. Voor een leeg element is er juist één alternatief, met name ε . Voor het niet-terminale element `film` uit het lopend voorbeeld is er slechts één alternatief, dat bestaat uit een opeenvolging van vier onderliggende elementen. We krijgen dus een tweedimensionale rij met dimensie (1×4) .

Merk op dat opeenvolgingen en alternatieven in een DTD vrij mogen genest worden, zodat ook voor het type `StructType` de mogelijkheid zal bestaan om (recursief) een opeenvolging of een lijst van alternatieven te modelleren. Dit recursieve datatype wordt gebruikt om de structuur van de individuele attributen en onderliggende elementen te modelleren:

$$\begin{aligned}
 \text{def StructType} := \text{Choice} & \left(\begin{aligned}
 & \text{enum} \mapsto (\mathbb{A}^+)^+, \\
 & \text{default} \mapsto \text{Record} (\text{type} \mapsto \text{StructType}, \text{value} \mapsto \mathbb{A}^*), \\
 & \text{maybe} \mapsto \text{StructType}, \\
 & \text{list} \mapsto \text{StructType}, \\
 & \text{tuple} \mapsto \text{StructType}^*, \\
 & \text{oneof} \mapsto \text{StructType}^+, \\
 & \text{string} \mapsto \iota \text{ string}, \\
 & \text{defined} \mapsto \mathbb{A}^+
 \end{aligned} \right) \tag{2.3}
 \end{aligned}$$

De gebruikte labels corresponderen met de verschillende syntactische DTD-constructies uit 1.1.2. Het label *enum* wordt gebruikt om een enumeratie te modelleren en met *default* kan een eventuele verstekwaarde opgegeven worden. De labels *maybe* en *list* duiden respectievelijk op een optioneel element (aangeduid met `?` in de DTD) en op de herhaling van een element (aangeduid door `*` of `+`). De labels *tuple* en *oneof* worden gebruikt om binnen een bepaalde waarde van het type `StructType` een opeenvolging (gescheiden door `,` in de DTD) of een lijst van alternatieven (gescheiden door `|`) te modelleren. Het label *string* wijst op het voorkomen van karakterdata in het document en *defined* kan gebruikt worden om door het opgeven van een naam een (andere) typedefinitie te selecteren, corresponderend met een nieuw XML-element.

2.2.2 Formele beschrijving van `DtdToHaskell`

Het parseren van geldige DTD-documenten gebeurt door het programma `DtdToHaskell`. We krijgen dan als resultaat een niet-lege rij typedefinities. We beschikken ook over een impliciete validiteitsfunctie, want het parseren lukt enkel voor geldige DTD-bestanden. Voor een document dat geen geldige DTD voorstelt, genereert de parser een foutmelding die de locatie en de aard van de eerste syntactische fout aangeeft. Voor onze doeleinden zijn enkel de types van de validiteitsfunctie en de parseerfunctie van belang:

$$\begin{aligned}
 \text{validdtd} & : \mathbb{A}^* \rightarrow \mathbb{B} \\
 \text{parsedtd} & : (\mathbb{A}^*)_{\text{validdtd}} \rightarrow \text{Typedef}^+
 \end{aligned}$$

De beeldpuntdefinitie van deze functies laten we hier buiten beschouwing. We zullen de HaXml toolkit conceptueel als een **black box** blijven behandelen. Voor onze voorbeeld-DTD krijgen we het volgende resultaat:

$$\begin{aligned} \text{validdtd } \text{filmdtd} &= 1 \\ \text{def } \text{filmdef} &:= \text{parsedtd } \text{filmdtd} \end{aligned} \quad (2.4)$$

Het resultaat van het parseren van deze DTD is een rij van zes typedefinities, die corresponderen met de zes specificatieregels in het oorspronkelijke DTD-bestand:

$$\begin{aligned} \text{filmdef } 0 &= \mathbf{n} \mapsto \text{“film”} \\ &\cup \mathbf{a} \mapsto \tau(\text{defined} \mapsto \text{“genre”}) \\ &\cup \mathbf{c} \mapsto \tau(\text{defined} \mapsto \text{“titel”,} \\ &\quad \text{defined} \mapsto \text{“regisseur”,} \\ &\quad \text{maybe} \mapsto \text{defined} \mapsto \text{“jaar”,} \\ &\quad \text{list} \mapsto \text{defined} \mapsto \text{“acteur”}) \\ \text{filmdef } 1 &= \mathbf{n} \mapsto \text{“genre”} \\ &\cup \mathbf{a} \mapsto \varepsilon \\ &\cup \mathbf{c} \mapsto \tau^2(\text{enum} \mapsto (\text{“Actie”, “Drama”, “Komedie”, “Thriller”})) \\ \text{filmdef } 2 &= \mathbf{n} \mapsto \text{“titel”} \\ &\cup \mathbf{a} \mapsto \varepsilon \\ &\cup \mathbf{c} \mapsto \tau^2(\text{string} \mapsto \text{string}) \\ \text{filmdef } 3 &= \mathbf{n} \mapsto \text{“regisseur”} \\ &\cup \mathbf{a} \mapsto \varepsilon \\ &\cup \mathbf{c} \mapsto \tau^2(\text{string} \mapsto \text{string}) \\ \text{filmdef } 4 &= \mathbf{n} \mapsto \text{“jaar”} \\ &\cup \mathbf{a} \mapsto \varepsilon \\ &\cup \mathbf{c} \mapsto \tau^2(\text{string} \mapsto \text{string}) \\ \text{filmdef } 5 &= \mathbf{n} \mapsto \text{“acteur”} \\ &\cup \mathbf{a} \mapsto \varepsilon \\ &\cup \mathbf{c} \mapsto \tau^2(\text{string} \mapsto \text{string}) \end{aligned}$$

Merk op dat de omzetting van een DTD naar een rij typedefinities enige redundantie kan invoeren. Formeel kunnen we stellen dat de functie `parsedtd` geen surjectie is van een teksbestand op een rij typedefinities. Zo kan men bijvoorbeeld uit bovenstaande typedefinities expliciet afleiden dat “genre” geen attributen heeft. Deze informatie is echter ook impliciet af te leiden uit het feit dat “genre” zelf een attribuut is. Dergelijke overtollige informatie zal echter toelaten de DTD-afhankelijke functionaliteit op een uniforme manier te beschrijven. In strikte zin is de functie `parsedtd` ook geen injectie, omdat verschillende tekstbestanden dezelfde DTD kunnen voorstellen en dus ook dezelfde typedefinities kunnen genereren.

We kunnen nu de DTD-afhankelijke functionaliteit modelleren door functies met als parameter een rij typedefinities. De verdere werking van het programma `DtdToHaskell` wordt gekarakteriseerd door de volgende vier functies:

$$\begin{aligned} \text{toType} & : \text{Typedef}^+ \rightarrow \mathcal{T} \\ \text{valid} & : \text{Typedef}^+ \rightarrow \mathbb{A}^* \rightarrow \mathbb{B} \\ \text{open}__ & : \text{Typedef}^+ \ni t \rightarrow (\mathbb{A}^*)_{\text{valid } t} \rightarrow \text{toType } t \\ \text{save}__ & : \text{Typedef}^+ \ni t \rightarrow \text{toType } t \rightarrow (\mathbb{A}^*)_{\text{valid } t} \end{aligned}$$

De functie `toType` zet een rij typedefinities om naar een eenvoudig recursief datatype. De instanties van dit datatype stellen dan precies die XML-documenten voor die grammaticaal correct zijn met betrekking tot de gegeven typedefinities en dus ook met betrekking tot de oorspronkelijke DTD.

De functie `open` is een parser voor dergelijke XML-documenten. Opnieuw beschikken we over een validiteitsfunctie, want voor tekstbestanden die niet zowel syntactisch als grammaticaal correct zijn, krijgen we een foutmelding.

De functie `save` is de semi-inverse van de functie `open`. Concreet betekent dit dat de samenstelling $(\text{open} \circ \text{save})$ gelijk is aan de identiteitsfunctie, maar dat dit niet noodzakelijk geldt voor de samenstelling $(\text{save} \circ \text{open})$. De XML syntax laat namelijk een zekere vrijheid toe, onder meer voor het gebruik van witruimte, zodat verschillende tekstbestanden hetzelfde XML-document kunnen voorstellen. Deze verschillen zullen echter wegvallen na het openen en opnieuw bewaren van deze bestanden, omdat het bewaren steeds op precies dezelfde manier gebeurt.

De interne werking van bovenstaande functies behoort tot de **black box** van het programma `DtdToHaskell`, maar bij wijze van illustratie beschrijven we nu de beeldpuntdefinitie van de functie `toType`. De aard van deze functie is namelijk uitermate geschikt om de structuur van

de datatypes `Typedef` en (vooral) `StructType` te verduidelijken. De functie `toType` heeft als enig argument een rij typedefinities en genereert in essentie een type voor het documentelement. We zullen echter twee hulpfuncties invoeren die een extra argument hebben. De eerste hulpfunctie `typedefToType` laat toe door middel van een index een bepaalde typedefinitie uit de rij te selecteren. Omdat we als conventie hebben aangenomen dat de typedefinitie voor het documentelement de eerste is in de rij, kunnen we voor het genereren van het corresponderende datatype de functie `typedefToType` met index 0 als argument gebruiken:

$$\mathbf{def\ toType: Typedef^+ \rightarrow \mathcal{T}\ with\ toType\ t = typedefToType\ t\ 0} \quad (2.5)$$

De functie `structToType` is de tweede hulpfunctie, met als argument een waarde van het type `StructType`. De functie `typedefToType` distribueert deze hulpfunctie over de attributen en de onderliggende elementen van een typedefinitie:

$$\begin{aligned} \mathbf{def\ typedefToType: Typedef^+ \ni t \rightarrow \mathcal{D}\ t \rightarrow \mathcal{T}} \\ \mathbf{with\ typedefToType\ ti = } & \quad n \mapsto \iota(t\ i\ n) \\ & \cup \quad a \mapsto \times(\mathbf{structToType}\ t\ o\ t\ i\ a) \\ & \cup \quad c \mapsto |j: \mathcal{D}(t\ i\ c). \times(\mathbf{structToType}\ t\ o\ t\ i\ c\ j) \end{aligned} \quad (2.6)$$

Het resulterende type is samengesteld uit de naam van het element, het product van de attributtypes en het product van de types van de onderliggende elementen. Voor de onderliggende elementen kunnen verschillende alternatieven bestaan. Deze keuze wordt met behulp van de `|`-operator gemodelleerd in het datatype.

De hulpfunctie `structToType` associeert met elke waarde van het type `StructType` een datatype, m.a.w. een verzameling mogelijke waarden. We kunnen deze functie dus beschouwen als een semantische functie voor DTD's. Zoals verwacht wordt het recursieve karakter van het type

StructType weerspiegelt in de onderstaande functie:

$$\begin{aligned}
& \mathbf{def} \text{ structToType} : \text{Typedef}^+ \ni t \rightarrow \text{StructType} \rightarrow \mathcal{T} \\
& \mathbf{with} \text{ structToType } t = (s : | (enum \mapsto (\mathbb{A}^+)^+) . \mathcal{R} (s \text{ enum})) \\
& \cup (s : | (default \mapsto \text{Record} (type \mapsto \text{StructType}, value \mapsto \mathbb{A}^*)) . \text{structToType } t (s \text{ default type})) \\
& \cup (s : | (maybe \mapsto \text{StructType}) . \iota \text{ Nothing} \cup \text{structToType } t (s \text{ maybe})) \\
& \cup (s : | (list \mapsto \text{StructType}) . (\text{structToType } t (s \text{ list}))^*) \\
& \cup (s : | (tuple \mapsto \text{StructType}^*) . \times (\text{structToType } t \circ s \text{ tuple})) \\
& \cup (s : | (oneof \mapsto \text{StructType}^+) . | (\text{structToType } t \circ s \text{ oneof})) \\
& \cup (s : | (string \mapsto \iota \text{ string}) . \mathbb{A}^+) \\
& \cup (s : | (defined \mapsto \mathbb{A}^+) . \text{typedefToType } t ((t^\top \mathbf{n})^- (s \text{ defined})))
\end{aligned} \tag{2.7}$$

Merk op hoe in de laatste regel de functie $(t^\top \mathbf{n})^-$ gebruikt wordt om de naam $(s \text{ defined})$ af te beelden op een index, die dan kan gebruikt worden als argument voor de functie `typedefToType`. Hierbij wordt verondersteld dat de naam $(s \text{ defined})$ precies eenmaal voorkomt in de rij type-definities. Dit vormt geen probleem omdat we werken met typedefinities die een geldige DTD voorstellen. In een dergelijke DTD zijn alle elementen waar naar verwezen wordt eenduidig gedefinieerd.

Stellen we nu bijvoorbeeld:

$$\mathbf{def} \text{ filmttype} := \text{toType } \text{filmdef}$$

Dan volgt rechtstreeks uit (2.5):

$$\text{filmttype} = \text{typedefToType } \text{filmdef } 0$$

Toepassing van (2.6) en (2.4) levert:

$$\begin{aligned}
\text{filmttype} = & \mathbf{n} \mapsto \iota \text{ "film"} \\
& \cup \mathbf{a} \mapsto \times (\text{structToType } \text{filmdef} \circ \tau (defined \mapsto \text{"genre"})) \\
& \cup \mathbf{c} \mapsto |j : \square 1 . \times (\text{structToType } \text{filmdef} \circ \tau (defined \mapsto \text{"titel"}, \\
& \hspace{15em} defined \mapsto \text{"regisseur"}, \\
& \hspace{15em} maybe \mapsto defined \mapsto \text{"jaar"}, \\
& \hspace{15em} list \mapsto defined \mapsto \text{"acteur"})) j)
\end{aligned} \tag{2.8}$$

Men kan gemakkelijk aantonen dat $f \circ \tau x = \tau(f x)$ voor $x : \mathcal{D} f$ en dat $\times(\tau y) = \square 1 \rightarrow y$. Er geldt dus ook dat $\times(f \circ \tau x) = \square 1 \rightarrow f x$, zodat we de tweede regel in (2.8) kunnen herschrijven als

$$\text{filmtype } a = \square 1 \rightarrow \text{structToType } \text{filmdef } (\text{defined} \mapsto \text{“genre”}). \quad (2.9)$$

Er geldt $\square 1 = \iota 0$ en eveneens $|j : \iota i . f j = \iota i \rightarrow f i$, zodat we de laatste regel in (2.8) kunnen herleiden tot

$$\begin{aligned} \text{filmtype } c = \square 1 \rightarrow \times(\text{structToType } \text{filmdef } \circ (\text{defined} \mapsto \text{“titel”}, \\ \text{defined} \mapsto \text{“regisseur”}, \\ \text{maybe} \mapsto \text{defined} \mapsto \text{“jaar”}, \\ \text{list} \mapsto \text{defined} \mapsto \text{“acteur”})). \end{aligned}$$

Als we nu de in hoofdstuk 1 vermelde eigenschappen $f \circ (x, y, z) = f x, f y, f z$ en $\times(A, B) = A \times B$ toepassen, vinden we:

$$\begin{aligned} \text{filmtype } c = \square 1 \rightarrow \text{structToType } \text{filmdef } (\text{defined} \mapsto \text{“titel”}) \\ \times \text{structToType } \text{filmdef } (\text{defined} \mapsto \text{“regisseur”}) \\ \times \text{structToType } \text{filmdef } (\text{maybe} \mapsto \text{defined} \mapsto \text{“jaar”}) \\ \times \text{structToType } \text{filmdef } (\text{list} \mapsto \text{defined} \mapsto \text{“acteur”}). \end{aligned} \quad (2.10)$$

Invullen van (2.9) en (2.10) in (2.8) en toepassing van (2.7) voor de labels *defined*, *maybe* en *list* levert

$$\begin{aligned} \text{filmtype} = \text{ n} \mapsto \iota \text{“film”} \\ \cup \text{ a} \mapsto \square 1 \rightarrow \text{typedefToType } \text{filmdef } 1 \\ \cup \text{ c} \mapsto \square 1 \rightarrow \text{typedefToType } \text{filmdef } 2 \\ \times \text{typedefToType } \text{filmdef } 3 \\ \times (\iota \text{Nothing} \cup \text{typedefToType } \text{filmdef } 4) \\ \times (\text{typedefToType } \text{filmdef } 5)^*. \end{aligned} \quad (2.11)$$

We kunnen nu opnieuw (2.6) toepassen op de overblijvende typedefinities. Wanneer we bijvoorbeeld de laatste regel uit (2.11) verder uitwerken vinden we op analoge manier, rekening houdend

met (2.4) en (2.7) :

$$\begin{aligned}
\text{typedefToType } \mathit{filmdef} \text{ } \mathfrak{S} &= \mathfrak{n} \mapsto \iota \text{“acteur”} \\
&\cup \mathfrak{a} \mapsto \times (\text{structToType } \mathit{filmdef} \circ \varepsilon) \\
&\cup \mathfrak{c} \mapsto |j : \square 1 . \times (\text{structToType } \mathit{filmdef} \circ \tau^2 (\text{string} \mapsto \text{string}) j) \\
&= \mathfrak{n} \mapsto \iota \text{“acteur”} \cup \mathfrak{a} \mapsto \iota \varepsilon \\
&\cup \mathfrak{c} \mapsto \square 1 \rightarrow \times (\text{structToType } \mathit{filmdef} \circ \tau (\text{string} \mapsto \text{string})) \\
&= \mathfrak{n} \mapsto \iota \text{“acteur”} \cup \mathfrak{a} \mapsto \iota \varepsilon \\
&\cup \mathfrak{c} \mapsto \square 1 \rightarrow \square 1 \rightarrow \text{structToType } \mathit{filmdef} (\text{string} \mapsto \text{string}) \\
&= \mathfrak{n} \mapsto \iota \text{“acteur”} \cup \mathfrak{a} \mapsto \iota \varepsilon \cup \mathfrak{c} \mapsto \square 1 \rightarrow \square 1 \rightarrow \mathbb{A}^+
\end{aligned}$$

Hierbij werd opnieuw gebruik gemaakt van de eigenschappen $|j : \iota i . f j = \iota i \rightarrow f i$ en $\times (\tau y) = \square 1 \rightarrow y$. In de laatste uitdrukking kunnen we $\square 1 \rightarrow \square 1 \rightarrow \mathbb{A}^+$ nog verkort noteren als $(\mathbb{A}^+)^{1,1}$. Wanneer we (2.11) op deze manier verder uitwerken vinden we uiteindelijk:

$$\begin{aligned}
\mathit{filmtype} &= \mathfrak{n} \mapsto \iota \text{“film”} && (2.12) \\
&\cup \mathfrak{a} \mapsto \square 1 \rightarrow \mathfrak{n} \mapsto \iota \text{“genre”} \\
&\quad \cup \mathfrak{a} \mapsto \iota \varepsilon \\
&\quad \cup \mathfrak{c} \mapsto \{\text{“Actie”, “Drama”, “Komedie”, “Thriller”}\}^{1,1} \\
&\cup \mathfrak{c} \mapsto \square 1 \rightarrow (\mathfrak{n} \mapsto \iota \text{“titel”} \cup \mathfrak{a} \mapsto \iota \varepsilon \cup \mathfrak{c} \mapsto (\mathbb{A}^+)^{1,1}) \\
&\quad \times (\mathfrak{n} \mapsto \iota \text{“regisseur”} \cup \mathfrak{a} \mapsto \iota \varepsilon \cup \mathfrak{c} \mapsto (\mathbb{A}^+)^{1,1}) \\
&\quad \times (\iota \text{Nothing} \cup (\mathfrak{n} \mapsto \iota \text{“jaar”} \cup \mathfrak{a} \mapsto \iota \varepsilon \cup \mathfrak{c} \mapsto (\mathbb{A}^+)^{1,1})) \\
&\quad \times (\mathfrak{n} \mapsto \iota \text{“acteur”} \cup \mathfrak{a} \mapsto \iota \varepsilon \cup \mathfrak{c} \mapsto (\mathbb{A}^+)^{1,1})^*
\end{aligned}$$

De bovenstaande uitdrukking is een datatype en dus een verzameling geldige waarden. Concreet betekent dit dat elk XML-document dat grammaticaal correct is met betrekking tot de oorspronkelijke DTD, eenduidig kan voorgesteld worden door een waarde van dit datatype. Omgekeerd kan men ook met elke waarde van dit datatype een XML-document associëren. Beschouwen bijvoorbeeld het volgende XML-document uit hoofdstuk 1 (opnieuw een tekstbestand van het

type \mathbb{A}^*):

```
def filmdoc := "<film genre='Actie'>
  <titel>The Untouchables</titel>
  <regisseur>Brian De Palma</regisseur>
  <jaar>1987</jaar>
  <acteur>Kevin Costner</acteur>
  <acteur>Sean Connery</acteur>
  <acteur>Robert De Niro</acteur>
</film>"
```

Het document *filmdoc* is syntactisch en grammaticaal correct en voldoet dus aan de specificaties van *filmdtd*. Formeel betekent dit:

$$\text{valid } \text{filmdef } \text{filmdoc} = 1 \quad \text{of m.a.w.} \quad \text{filmdoc} \in (\mathbb{A}^*)_{\text{valid filmdef}}$$

De met *filmdoc* corresponderende waarde van het datatype *filmtype* wordt dan gegeven door toepassing van de functie *open*:

$$\begin{aligned} \text{def } \text{filmval} &:= \text{open}_{\text{filmdef}} \text{filmdoc} \\ \text{filmval} &= \text{n} \mapsto \text{"film"} \\ &\cup \text{a} \mapsto \tau (\text{n} \mapsto \text{"genre"} \cup \text{a} \mapsto \varepsilon \cup \text{c} \mapsto \tau^2 \text{"Actie"}) \\ &\cup \text{c} \mapsto \tau (\text{n} \mapsto \text{"titel"} \cup \text{a} \mapsto \varepsilon \cup \text{c} \mapsto \tau^2 \text{"The Untouchables"}, \\ &\quad \text{n} \mapsto \text{"regisseur"} \cup \text{a} \mapsto \varepsilon \cup \text{c} \mapsto \tau^2 \text{"Brian De Palma"}, \\ &\quad \text{n} \mapsto \text{"jaar"} \cup \text{a} \mapsto \varepsilon \cup \text{c} \mapsto \tau^2 \text{"1987"}, \\ &\quad (\text{n} \mapsto \text{"acteur"} \cup \text{a} \mapsto \varepsilon \cup \text{c} \mapsto \tau^2 \text{"Kevin Costner"}, \\ &\quad \text{n} \mapsto \text{"acteur"} \cup \text{a} \mapsto \varepsilon \cup \text{c} \mapsto \tau^2 \text{"Sean Connery"}, \\ &\quad \text{n} \mapsto \text{"acteur"} \cup \text{a} \mapsto \varepsilon \cup \text{c} \mapsto \tau^2 \text{"Robert De Niro"})) \end{aligned} \tag{2.13}$$

Merk op dat, hoewel het datatype *filmtype* behoorlijk complex kan lijken, de bovenstaande waarde van dit datatype vrij eenvoudig te interpreteren valt en bovendien een sterk uniforme structuur vertoont, wat de verwerking van dergelijke waarden zal vereenvoudigen.

2.2.3 Formele beschrijving van de editor-functionaliteit

Tot zover hebben we de functionaliteit van het programma `DtdToHaskell` (gedeeltelijk) gemodelleerd in `Funmath`. In wat volgt zullen we de eigenlijke editor-functionaliteit modelleren. Het resultaat hiervan zal een basis vormen voor een uiteindelijke implementatie in Haskell. Vooreerst is er de functie `new`, die een verstekwaarde van een gegeven datatype construeert. Het XML-document dat correspondeert met deze verstekwaarde is het minimale document dat voldoet aan de met het datatype geassocieerde DTD. De functie `new` zal dan ook in de editor gebruikt worden bij het creëren van een nieuw document. Zoals alle andere DTD-afhankelijke functies heeft deze functie als argument een rij typedefinities. Niet enkel de waarde maar ook het type van het resultaat is afhankelijk van deze typedefinities:

$$\mathbf{def\ new_} : \text{Typedef}^+ \ni t \rightarrow \text{toType } t \mathbf{\ with\ new}_t = \text{newTypedef } t\ 0 \quad (2.14)$$

Net zoals bij de functie `toType` maken we opnieuw gebruik van twee hulpfuncties, namelijk `newTypedef` en `newStruct`, die als extra argument respectievelijk een index in de rij typedefinities en een waarde van het type `StructType` hebben. De functie `newTypedef` distribueert de functie `StructType` over de attributen en onderliggende elementen. Indien er meerdere alternatieven zijn voor de samenstelling van de onderliggende elementen, wordt het eerste alternatief gekozen:

$$\begin{aligned} \mathbf{def\ newTypedef} : \text{Typedef}^+ \ni t \rightarrow \mathcal{D}t \ni i \rightarrow \text{typedefToType } t\ i \\ \mathbf{with\ newTypedef } t\ i = \quad & n \mapsto (t\ i\ n) \\ & \cup \quad a \mapsto (\text{newStruct } t \circ t\ i\ a) \\ & \cup \quad c \mapsto \tau (\text{newStruct } t \circ t\ i\ c\ 0) \end{aligned} \quad (2.15)$$

De recursieve functie `newStruct` associeert een verstekwaarde met de elementaire datatypes zoals lijsten en karakterdata:

$$\begin{aligned}
& \mathbf{def} \text{ newStruct} : \text{Typedef}^+ \ni t \rightarrow \text{StructType} \ni s \rightarrow \text{structToType } t \\
& \mathbf{with} \text{ newStruct } t s = (s : | (enum \mapsto (\mathbb{A}^+)^+) . s \text{ enum } 0) \\
& \quad \cup (s : | (default \mapsto \text{Record } (type \mapsto \text{StructType}, value \mapsto \mathbb{A}^*)) . s \text{ default } value) \\
& \quad \cup (s : | (maybe \mapsto \text{StructType}) . \text{Nothing}) \\
& \quad \cup (s : | (list \mapsto \text{StructType}) . \varepsilon) \\
& \quad \cup (s : | (tuple \mapsto \text{StructType}^*) . \text{newStruct } t \circ s \text{ tuple}) \\
& \quad \cup (s : | (oneof \mapsto \text{StructType}^+) . (\text{newStruct } t \circ s \text{ oneof})] \iota 0) \\
& \quad \cup (s : | (string \mapsto \iota \text{ string}) . \text{"-"})) \\
& \quad \cup (s : | (defined \mapsto \mathbb{A}^+) . \text{newTypedef } t ((t^T \mathbf{n})^- (s \text{ defined})))
\end{aligned} \tag{2.16}$$

Voor de labels *default*, *maybe* en *list* is de verstekwaarde voor de hand liggend. Bij de labels *enum*, *oneof* en *string* wordt daarentegen (noodzakelijkerwijs) een vrij arbitraire keuze gemaakt. Door rechtstreekse toepassing van de definities bekomt men voor de voorbeeld-DTD achtereenvolgens:

$$\begin{aligned}
\text{new}_{\text{filmdef}} &= \langle (2.14) \rangle & \text{newTypedef } \text{filmdef } 0 \\
&= \langle (2.15), (2.4) \rangle & \mathbf{n} \mapsto \text{"film"} \\
& & \cup \mathbf{a} \mapsto (\text{newStruct } t \circ \tau (\text{defined} \mapsto \text{"genre"})) \\
& & \cup \mathbf{c} \mapsto \tau (\text{newStruct } t \circ (\text{defined} \mapsto \text{"titel"}, \\
& & \quad \text{defined} \mapsto \text{"regisseur"}, \\
& & \quad \text{maybe} \mapsto \text{defined} \mapsto \text{"jaar"}, \\
& & \quad \text{list} \mapsto \text{defined} \mapsto \text{"acteur"})) \\
&= \langle (2.16) \rangle & \mathbf{n} \mapsto \text{"film"} \\
& & \cup \mathbf{a} \mapsto \tau (\text{newTypedef } \text{filmdef } 1) \\
& & \cup \mathbf{c} \mapsto \tau (\text{newTypedef } \text{filmdef } 2, \\
& & \quad \text{newTypedef } \text{filmdef } 3, \\
& & \quad \text{Nothing}, \\
& & \quad \varepsilon)
\end{aligned}$$

Wanneer we deze uitdrukking op analoge wijze verder uitwerken, vinden we:

$$\begin{aligned}
\text{new}_{\text{filmdef}} = & \text{ n} \mapsto \text{“film”} \\
& \cup \text{ a} \mapsto \tau (\text{ n} \mapsto \text{“genre”} \cup \text{ a} \mapsto \varepsilon \cup \text{ c} \mapsto \tau^2 \text{“Actie”}) \\
& \cup \text{ c} \mapsto \tau (\text{ n} \mapsto \text{“titel”} \cup \text{ a} \mapsto \varepsilon \cup \text{ c} \mapsto \tau^2 \text{“-”}, \\
& \qquad \text{ n} \mapsto \text{“regisseur”} \cup \text{ a} \mapsto \varepsilon \cup \text{ c} \mapsto \tau^2 \text{“-”}, \\
& \qquad \text{Nothing}, \\
& \varepsilon)
\end{aligned} \tag{2.17}$$

Men kan gemakkelijk verifiëren dat uitdrukking (2.17) het type *filmtype* heeft en correspondeert met het onderstaande XML-document, dat bekomen wordt door toepassing van de functie `save`:

```

savefilmdef newfilmdef := “<film genre=“Actie”>
    <titel>-</titel>
    <regisseur>-</regisseur>
</film>”

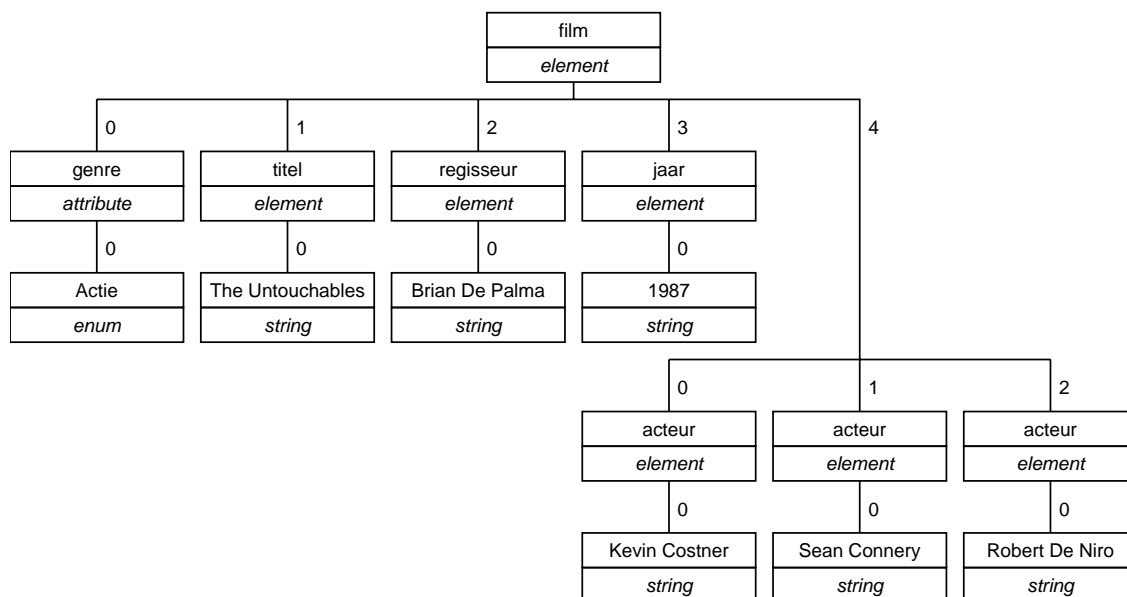
```

Bij het openen van een bestaand document of het creëren van een nieuw document wordt, zoals in de voorgaande voorbeelden, door de editor een waarde van een bepaald datatype gecreëerd voor interne verwerking. Aangezien de editor de gebruiker in staat moet stellen het document te wijzigen, moet deze waarde in eerste instantie aan de gebruiker gepresenteerd worden, bij voorkeur op het scherm. In principe kan een willekeurige representatie gekozen worden voor het weergeven van een XML-document. Zo kan men bijvoorbeeld opteren voor een grafische boomstructuur of voor een wiskundige vorm. Men zou ook de syntax van de programmeertaal kunnen gebruiken en rechtstreeks de interne representatie weergeven. De meest voor de hand liggende keuze is echter de XML syntax zelf, omdat de gebruiker er reeds mee vertrouwd is. In principe kunnen we de functie `save` gebruiken om een XML-bestand te genereren en de resulterende tekststring rechtstreeks op het scherm weergeven. Deze aanpak is echter weinig flexibel en staat haaks op het structurele aspect van de editor. We wensen namelijk de zuivere tekst te voorzien van extra informatie. Deze informatie kan dan gebruikt worden om de interactie met de gebruiker te vereenvoudigen. Ook kunnen bepaalde syntactische constructies op een speciale manier weergegeven worden om zo bijvoorbeeld **syntax highlighting** te realiseren. Om deze doelstellingen te verwezenlijken, kunnen we het XML-bestand opsplitsen in een rij tekststrings, corresponderend met de syntactische basisconstructies, en aan elke string bijkomende informatie

toevoegen. Met het oog op de opmaak van de tekst, associëren we vooreerst met elke deelstring een `ContentType`:

$$\text{def ContentType} := \{ \textit{element}, \textit{attribute}, \textit{enum}, \textit{string} \} \quad (2.18)$$

De verschillende labels van het type `ContentType` kunnen gebruikt worden om de syntactische basisconstructies van een XML-document op te splitsen in vier klassen: elementen, attributen, enumeraties en karakterdata. Deze indeling zou in principe verfijnd kunnen worden maar volstaat in deze vorm voor een basisimplementatie van syntax highlighting. Om de interactie met de gebruiker te realiseren, kunnen we verder aan elke string een unieke code verbinden. Deze code kan gebruikt worden om te navigeren in het document en om aan te geven op welke positie de **cursor** zich bevindt. Voor het formaat van de positiecode baseren we ons op de hiërarchische structuur van een XML-document. We kunnen een document namelijk voorstellen door een boomstructuur en de n opvolgers van elke knoop merken met de getallen 0 tot $n - 1$. De unieke code wordt dan voor elk element gegeven door de rij getallen die het pad naar het element specificeert. De interne representatie van het filmdocument staat afgebeeld in Fig. 2.1. Merk op



Figuur 2.1: Hiërarchische voorstelling van een XML-document

dat deze boomstructuur verschilt van de elementenboom uit Fig. 1.2. Hier wordt omwille van de eenvormigheid geen fundamenteel onderscheid gemaakt tussen attributen en onderliggende elementen. Anderzijds valt uit Fig. 2.1 wel expliciet af te leiden dat de verschillende acteurs een lijst vormen en dus geen onafhankelijke elementen zijn. Algemeen kunnen we stellen dat de

elementenboom sterk aanleunt bij de concrete XML syntax, terwijl de interne representatie zich situeert op een abstracter niveau en louter gebaseerd is op het onderliggende datatype.

De functie `showXml` neemt als argumenten een rij typedefinities en een waarde van het bijhorende datatype, en geeft als resultaat een representatie van die waarde onder de vorm van een rij tupels. Elk tupel bestaat uit een tekststring, een positiecode en een `ContentType`:

$$\begin{aligned} \mathbf{def} \text{ showXml}_- : \text{Typedef}^+ \ni t \rightarrow \text{toType } t \rightarrow (\mathbb{A}^* \times \mathbb{N}^* \times \text{ContentType})^+ \\ \mathbf{with} \text{ showXml}_t d = \text{showTypedef } \text{element } t \ 0 \ d \ \varepsilon \end{aligned} \quad (2.19)$$

Het eerste argument van de hulpfunctie `showTypedef` geeft aan of de geselecteerde typedefinitie een element of een attribuut karakteriseert. Deze informatie wordt gebruikt om een keuze te maken tussen de labels `element` en `attribute` voor de waarde van het type `ContentType`. Het tweede argument is een rij typedefinities en het derde argument een index in deze rij. Het vierde argument is de eigenlijke waarde van het attribuut of element en het laatste argument is een accumulator en wordt gebruikt bij het opbouwen van de positiecode. In essentie distribueert de functie `showTypedef` een hulpfunctie `showStruct` over de verschillende attributen en onderliggende elementen, en worden de resultaten geconcateneerd:

$$\begin{aligned} \mathbf{def} \text{ showTypedef} : \{ \text{element}, \text{attribute} \} \rightarrow \text{Typedef}^+ \ni t \rightarrow \mathcal{D} \ t \ni i \rightarrow \\ \text{typedefToType } t \ i \rightarrow \mathbb{N}^* \rightarrow (\mathbb{A}^* \times \mathbb{N}^* \times \text{ContentType})^+ \\ \mathbf{with} \text{ showTypedef } \ c \ t \ i \ d \ p = (d \ n, p, c) \succ \\ \begin{aligned} & \text{++} [\text{cart}(\text{showStruct } \text{attribute } t) \prec (t \ i \ a, d \ a, p \overleftarrow{\prec} (0 \dots \#(d \ a)))] \text{ ++} \\ & \text{++} [\text{cart}(\text{showStruct } \text{element } t) \prec (t \ i \ c \ j, d \ c \ j, p \overleftarrow{\prec} (\#(d \ a) \dots \#(d \ a) + \#(d \ c \ j))) \\ & \quad \mathbf{where } j := \iota^-(\mathcal{D}(d \ c))] \end{aligned} \end{aligned} \quad (2.20)$$

De concatenatie-operator `++` wordt gedefinieerd door het domein-axioma $\#(x ++ y) = \#x + \#y$ en het beeldpunt-axioma $\forall i : \mathcal{D}(x ++ y) . (x ++ y) i = (i < \#x) ? x i \uparrow y (i - \#x)$. De operatoren `>` en `<` worden gedefinieerd door $a > x = \tau a ++ x$ en $x < a = x ++ \tau a$. We nemen hier als conventie aan dat de functies `>` en `<` een hogere prioriteit krijgen dan de functie `++`, we stellen dus $a > x ++ y := (a > x) ++ y$. Er geldt echter $(a > x) ++ y = a > (x ++ y)$, zodat de conventie $a > x ++ y := a > (x ++ y)$ tot dezelfde interpretatie leidt van bovenstaande uitdrukking. Deze wederzijdse associativiteit vinden we ook terug in uitdrukkingen van de vorm $x ++ y < a$ en $a > x < b$. De operator `++` is een elastische versie van `++` en wordt als volgt

gedefinieerd:

$$\begin{aligned} \text{++} & : (A^*)^* \rightarrow A^* \\ \text{++ } \varepsilon & = \varepsilon \\ \text{++ } (x \succ xs) & = x \text{++ } \text{++ } xs \end{aligned}$$

De functie `cart` zet een functie in Curry-vorm met n argumenten om naar een functie in Cartesiaanse vorm met als enig argument een n -tupel:

$$\begin{aligned} \text{cart} & : (S \rightarrow R) \rightarrow \times S \rightarrow R \\ \text{cart } f \varepsilon & = f \\ \text{cart } f (a \succ x) & = \text{cart } (f a) x \end{aligned}$$

Hierin is S een willekeurige rij types en R een willekeurig type. In feite is `cart` polymorf met betrekking tot S en R , maar dat werd niet weergegeven in de typedefinitie omdat het invoeren en bespreken van de vereiste polymorfisme-operator ons te ver zou leiden.

Zoals vermeld in hoofdstuk 1 vormt de \prec -operator een combinatie van functiesamenstelling en functietranspositie, hier gebruikt om de argumenten voor de functie `showStruct` te distribueren. Merk op hoe voor het laatste argument telkens de positiecode wordt opgebouwd door aan de reeds geaccumuleerde rij een unieke index toe te voegen voor elk attribuut en elk onderliggend element. Indien er bijvoorbeeld twee attributen zijn, dan vinden we voor $p = \varepsilon$:

$$\begin{aligned} & (\text{cart } (\text{showStruct } \textit{attribute } t))^\prec ((t \textit{ i a } 0, t \textit{ i a } 1), (d \textit{ a } 0, d \textit{ a } 1), (\tau 0, \tau 1)) \\ = & \langle \text{definitie } \prec \rangle \quad \text{cart } (\text{showStruct } \textit{attribute } t) \circ ((t \textit{ i a } 0, t \textit{ i a } 1), (d \textit{ a } 0, d \textit{ a } 1), (\tau 0, \tau 1))^\top \\ = & \langle \text{definitie } \top \rangle \quad \text{cart } (\text{showStruct } \textit{attribute } t) \circ ((t \textit{ i a } 0, d \textit{ a } 0, \tau 0), (t \textit{ i a } 1, d \textit{ a } 1, \tau 1)) \\ = & \langle \text{definitie } \circ \rangle \quad (\text{cart } (\text{showStruct } \textit{attribute } t)(t \textit{ i a } 0, d \textit{ a } 0, \tau 0), \\ & \quad \text{cart } (\text{showStruct } \textit{attribute } t)(t \textit{ i a } 1, d \textit{ a } 1, \tau 1)) \\ = & \langle \text{definitie } \text{cart} \rangle \quad (\text{showStruct } \textit{attribute } t (t \textit{ i a } 0) (d \textit{ a } 0) (\tau 0), \\ & \quad \text{showStruct } \textit{attribute } t (t \textit{ i a } 1) (d \textit{ a } 1) (\tau 1)) \end{aligned} \tag{2.21}$$

De functie `showStruct` heeft gelijkaardige argumenten als de functie `showTypedef` maar heeft als

derde argument een waarde van het type `StructType`:

$$\begin{aligned}
& \mathbf{def} \text{ showStruct} : \{ \text{element}, \text{attribute} \} \rightarrow \text{Typedef}^+ \ni t \rightarrow \text{StructType} \ni s \rightarrow \\
& \quad \text{structToType } t \ s \rightarrow \mathbb{N}^* \rightarrow (\mathbb{A}^* \times \mathbb{N}^* \times \text{ContentType})^* \\
& \mathbf{with} \text{ showStruct } c \ t \ s \ d \ p = \phi \ s \ \mathbf{where} \\
& \phi = (s : | (enum \mapsto (\mathbb{A}^+)^+) . \tau(d, p, enum)) \\
& \quad \cup (s : | (default \mapsto \text{Record}(type \mapsto \text{StructType}, value \mapsto \mathbb{A}^*)) . \text{showStruct } c \ t \ (s \ \text{default } type) \ d \ p) \\
& \quad \cup (s : | (maybe \mapsto \text{StructType}) . (d = \text{Nothing}) ? \varepsilon \uparrow \text{showStruct } c \ t \ (s \ \text{maybe}) \ d \ p) \\
& \quad \cup (s : | (list \mapsto \text{StructType}) . \text{++} ((\text{cart}(\text{showStruct } c \ t \ (s \ \text{list})))^< (d, p \xrightarrow{\leftarrow} (0 \dots \# d)))) \\
& \quad \cup (s : | (tuple \mapsto \text{StructType}^*) . \text{++} ((\text{cart}(\text{showStruct } c \ t))^{< (s \ \text{tuple}, d, p \xrightarrow{\leftarrow} (0 \dots \# d)))) \\
& \quad \cup (s : | (oneof \mapsto \text{StructType}^+) . \text{showStruct } c \ t \ (s \ \text{oneof } j) \ (d \ j) \ p \ \mathbf{where} \ j = \iota^-(\mathcal{D} \ d)) \\
& \quad \cup (s : | (string \mapsto \iota \ \text{string}) . \tau(d, p, string)) \\
& \quad \cup (s : | (defined \mapsto \mathbb{A}^+) . \text{showTypedef } c \ t \ ((t^\top \ n)^- (s \ \text{defined})) \ d \ p)
\end{aligned} \tag{2.22}$$

Voor ons lopend voorbeeld kunnen we de functie `showXml` als volgt toepassen:

$$\begin{aligned}
& \text{showXml}_{\text{filmdef}} \ \text{filmval} \\
& = \langle (2.19) \rangle \quad \text{showTypedef } \text{element} \ \text{filmdef} \ 0 \ \text{filmval} \ \varepsilon \\
& = \langle (2.20), (2.13) \rangle \quad (\text{“film”}, \varepsilon, \text{element}) \succ - \\
& \text{++} ((\text{cart}(\text{showStruct } \text{attribute} \ \text{filmdef}))^{< (\text{filmdef } 0 \ \mathbf{a}, \text{filmval } \mathbf{a}, \varepsilon \xrightarrow{\leftarrow} (0 \dots 1))}) \text{++} \\
& \text{++} ((\text{cart}(\text{showStruct } \text{element} \ \text{filmdef}))^{< (\text{filmdef } 0 \ \mathbf{c} \ 0, \text{filmval } \mathbf{c} \ 0, \varepsilon \xrightarrow{\leftarrow} (1 \dots 1 + 4))})
\end{aligned}$$

Wanneer we bijvoorbeeld het gedeelte van de attributen verder uitwerken, vinden we:

$$\begin{aligned}
& (\text{cart}(\text{showStruct } \text{attribute} \ \text{filmdef}))^{< (\text{filmdef } 0 \ \mathbf{a}, \text{filmval } \mathbf{a}, \varepsilon \xrightarrow{\leftarrow} (0 \dots 1))} \\
& = \langle \dots, \rightarrow, \leftarrow \rangle \quad (\text{cart}(\text{showStruct } \text{attribute} \ \text{filmdef}))^{< (\text{filmdef } 0 \ \mathbf{a}, \text{filmval } \mathbf{a}, \tau \ 0)} \\
& = \langle \text{cfr. (2.21)} \rangle \quad \tau(\text{showStruct } \text{attribute} \ \text{filmdef} \ (\text{filmdef } 0 \ \mathbf{a}) \ (\text{filmval } \mathbf{a}) \ (\tau \ 0)) \\
& = \langle (2.4), (2.13) \rangle \quad \tau(\text{showStruct } \text{attribute} \ \text{filmdef} \ (\tau(\text{defined} \mapsto \text{“genre”})) \\
& \quad \quad \quad (\tau(\mathbf{n} \mapsto \text{“genre”} \cup \mathbf{a} \mapsto \varepsilon \cup \mathbf{c} \mapsto \tau^2 \text{“Actie”})) \ (\tau \ 0)) \\
& = \langle (2.22) \rangle \quad \text{showTypedef } \text{attribute} \ \text{filmdef} \ 1 \\
& \quad \quad \quad (\tau(\mathbf{n} \mapsto \text{“genre”} \cup \mathbf{a} \mapsto \varepsilon \cup \mathbf{c} \mapsto \tau^2 \text{“Actie”})) \ (\tau \ 0)
\end{aligned}$$

Verdere uitwerking van deze laatste uitdrukking leidt uiteindelijk tot:

$$\begin{aligned} & (\text{"genre"}, \tau 0, \textit{attribute}) \succ - ((\text{++} \ \varepsilon) \text{++} (\text{++} \ \tau (\text{"Actie"}, (0, 0), \textit{enum}))) \\ = & (\text{"genre"}, \tau 0, \textit{attribute}), (\text{"Actie"}, (0, 0), \textit{enum})) \end{aligned}$$

Op analoge manier kunnen ook de onderliggende elementen behandeld worden, we krijgen dan als resultaat:

$$\begin{aligned} \text{showXml}_{\textit{filmdef}} \ \textit{filmval} = & (\text{"film"}, \varepsilon, \textit{element}), \\ & (\text{"genre"}, \tau 0, \textit{attribute}), (\text{"Actie"}, (0, 0), \textit{enum}), \\ & (\text{"titel"}, \tau 1, \textit{element}), (\text{"The Untouchables"}, (1, 0), \textit{string}), \\ & (\text{"regisseur"}, \tau 2, \textit{element}), (\text{"Brian De Palma"}, (2, 0), \textit{string}), \\ & (\text{"jaar"}, \tau 3, \textit{element}), (\text{"1987"}, (3, 0), \textit{string}), \\ & (\text{"acteur"}, (4, 0), \textit{element}), (\text{"Kevin Costner"}, (4, 0, 0), \textit{string}), \\ & (\text{"acteur"}, (4, 1), \textit{element}), (\text{"Sean Connery"}, (4, 1, 0), \textit{string}), \\ & (\text{"acteur"}, (4, 2), \textit{element}), (\text{"Robert De Niro"}, (4, 2, 0), \textit{string})) \end{aligned}$$

Het resultaat van toepassing van de functie `showXml` op een concrete waarde blijkt eenvoudiger te interpreteren dan de functiedefinitie. In bovenstaande structuur vinden we duidelijk alle inhoud van het oorspronkelijke XML-document terug. Uit de positiecodes kunnen we eenduidig de hiërarchische boomstructuur uit Fig. 2.1 afleiden. Dankzij de toegevoegde labels van het type `ContentType` kunnen we bepaalde categoriën van tekststrings bijvoorbeeld in een verschillende kleur weergeven op het scherm. Merk op dat de aaneenschakeling van de tekststrings uit bovenstaande waarde nog niet voldoet aan de XML syntaxregels. Dankzij de extra toegevoegde informatie kan deze syntax echter eenvoudig gereconstrueerd worden. Uiteindelijk krijgt de gebruiker dus een opgemaakt XML-document op het scherm te zien.

Wanneer de gebruiker een bepaald stuk tekst aanklikt, dan is het de bedoeling de mogelijke wijzigingen op die positie in het document weer te geven, bijvoorbeeld in een menu. De positie in het document is uniek bepaald door de positiecode geassocieerd met de aangeklikte tekststring. Deze positiecode kan dus gebruikt worden als argument voor de functie `alternatives`, die de mogelijke wijzigingen als resultaat terug geeft. Voor de eenvoud zullen we deze alternatieven voorstellen als tekststrings van het type \mathbb{A}^* , die rechtstreeks in een menu kunnen getoond worden. We introduceren eerst een hulpfunctie `getName`, die een waarde van het type `StructType` omzet

naar een voor de gebruiker eenvoudig te interpreteren stringrepresentatie:

```

def getName : StructType →  $\mathbb{A}^*$ 
with getName = (s : |(enum ↦ ( $\mathbb{A}^+$ )+) . par (insert ‘|’ (s enum)))
  ∪ (s : |(default ↦ Record (type ↦ StructType, value ↦  $\mathbb{A}^*$ )) . getName (s default type))
  ∪ (s : |(maybe ↦ StructType) . getName (s maybe) < ‘?’)
  ∪ (s : |(list ↦ StructType) . getName (s list) < ‘*’)
  ∪ (s : |(tuple ↦ StructType*) . par (insert ‘,’ (getName ∘ (s tuple))))
  ∪ (s : |(oneof ↦ StructType+) . par (insert ‘|’ (getName ∘ (s oneof))))
  ∪ (s : |(string ↦  $\iota$  string) . “String”)
  ∪ (s : |(defined ↦  $\mathbb{A}^+$ ) . s defined)

```

(2.23)

In bovenstaande definitie worden volgende hulpfuncties gebruikt:

$\text{par} : \mathbb{A}^* \rightarrow \mathbb{A}^*$	$\text{insert} : (A^*)^* \rightarrow A \rightarrow A^*$
$\text{par } s = ‘(? \succ s \prec ’$	$\text{insert } a \varepsilon = \varepsilon$
	$\text{insert } a (x \succ xs) = x ++ \text{++} (a \overset{\rightarrow}{\succ} xs)$

Om een zinvolle stringvoorstelling te bepalen voor een opeenvolging van waarden van het type StructType gebruiken we de functie getNames:

```

def getNames : StructType* →  $\mathbb{A}^*$ 
with getNames s = insert ‘_’ (getName ∘ s)

```

(2.24)

Er geldt bijvoorbeeld:

$$\text{getNames} (\text{filmdef } 0 \text{ c } 0) = \text{“titel regisseur jaar? acteur*”}$$

De functie alternatives neemt als argumenten een rij typedefinities, een document en een positie in dat document, en geeft als resultaat een rij tekststrings:

```

def alternatives— : Typedef+ ∋ t → toType t →  $\mathbb{N}^*$  → ( $\mathbb{A}^+$ )*
with alternativest d p = altTypedef t 0 d p

```

(2.25)

Zoals steeds zijn er twee hulpfuncties, met als argument een index die de typedefinitie bepaalt of een waarde van het type StructType. De positiecode wordt in deze functies gebruikt om het hiërarchisch gestructureerde document te ontmantelen en zo de positie te bereiken die door de

gebruiker aangeduid werd. Bij het afdalen in deze boomstructuur wordt de lijst van mogelijke wijzigingen aan het document opgebouwd. De functie `altTypedef` wordt als volgt gedefinieerd:

$$\begin{aligned}
& \mathbf{def} \text{ altTypedef} : \text{Typedef}^+ \ni t \rightarrow \mathcal{D} t \ni i \rightarrow \text{typedefToType } t i \rightarrow \mathbb{N}^* \rightarrow (\mathbb{A}^+)^* \\
& \mathbf{with} \text{ altTypedef } t i d p = (p = \varepsilon) ? (\text{“Change_To_”} \overrightarrow{++} (\text{getNames} \circ t i c)) \dagger \phi (p 0) \\
& \mathbf{where} \phi m = (m < \#(t i a)) ? \mathbf{altStruct } t (t i a m) (d a m) (\sigma p) \dagger \psi (m - \#(t i a)) \\
& \mathbf{where} \psi n = \mathbf{altStruct } t (t i c j n) (d c j n) (\sigma p) \mathbf{where} j = \iota^-(\mathcal{D} (d c))
\end{aligned} \tag{2.26}$$

Indien de resterende positiecode leeg is ($p = \varepsilon$), dan werd het huidige element zelf door de gebruiker geselecteerd en bestaat de enige mogelijke wijziging uit het kiezen van een nieuw alternatief voor de structuur van de onderliggende elementen. Deze alternatieven worden gegeven door $t i c$. Als de resterende positiecode niet leeg is, dan zal de functie `altTypedef` bepalen welk attribuut of onderliggend element van het huidige element door de gebruiker geselecteerd werd aan de hand van de index $p 0$. Het geselecteerde attribuut of onderliggend element wordt dan meegegeven als argument aan de functie `altStruct`. Deze hulpfunctie zal het document verder ontmantelen en de eigenlijke alternatieven bepalen:

$$\begin{aligned}
& \mathbf{def} \text{ altStruct} : \text{Typedef}^+ \ni t \rightarrow \text{StructType} \ni s \rightarrow \text{structToType } t s \rightarrow \mathbb{N}^* \rightarrow (\mathbb{A}^+)^* \\
& \mathbf{with} \text{ altStruct } t s d p = \phi s \mathbf{where} \\
& \phi = (s : |(\text{enum} \mapsto (\mathbb{A}^+)^+). \text{“Change_To_”} \overrightarrow{++} (s \text{ enum})) \\
& \quad \cup (s : |(\text{default} \mapsto \text{Record } (type \mapsto \text{StructType}, value \mapsto \mathbb{A}^*)). \mathbf{altStruct } t (s \text{ default type}) d p) \\
& \quad \cup (s : |(\text{maybe} \mapsto \text{StructType}). d = \text{Nothing} ? \tau (\text{“Add_”} \overrightarrow{++} \text{getName } (s \text{ maybe})) \\
& \quad \quad \dagger (\text{“Delete_”} \overrightarrow{++} \text{getName } (s \text{ maybe})) \succ \mathbf{altStruct } t (s \text{ maybe}) d p) \\
& \quad \cup (s : |(\text{list} \mapsto \text{StructType}). (\text{“Add_”} \overrightarrow{++} \text{getName } (s \text{ list})) \succ \\
& \quad \quad (d = \varepsilon ? \varepsilon \dagger (\text{“Delete_”} \overrightarrow{++} \text{getName } (s \text{ list})) \succ \mathbf{altStruct } t (s \text{ list}) (d (p 0)) (\sigma p))) \\
& \quad \cup (s : |(\text{tuple} \mapsto \text{StructType}^*). \mathbf{altStruct } t (s \text{ tuple } (p 0)) (d (p 0)) (\sigma p)) \\
& \quad \cup (s : |(\text{oneof} \mapsto \text{StructType}^+). (\text{“Change_To_”} \overrightarrow{++} (\text{getName} \circ s \text{ oneof})) \overrightarrow{++} \\
& \quad \quad \mathbf{altStruct } t (s \text{ oneof } j) (d j) p \mathbf{where} j = \iota^-(\mathcal{D} d) \\
& \quad \cup (s : |(\text{string} \mapsto \iota \text{ string}). \tau \text{“Edit_String”} \\
& \quad \cup (s : |(\text{defined} \mapsto \mathbb{A}^+). \mathbf{altTypedef } t ((t^\top n)^-(s \text{ defined})) d p
\end{aligned} \tag{2.27}$$

Bij wijze van voorbeeld bespreken we het label `list`: als de gebruiker een lijst in het document selecteert, dan is het toevoegen van een nieuw element aan de lijst steeds een geldige wijziging.

Indien de lijst niet leeg is, dan werd de lijst geselecteerd door het aanklikken van een bepaald element uit de lijst. Bijkomende mogelijke wijzigingen zijn in dat geval het verwijderen en het wijzigen van het geselecteerde element. Voor deze laatste optie kunnen opnieuw verschillende alternatieven bestaan, die bepaald worden door een recursieve oproep van de functie `altStruct`. Toegepast op het filmdocument krijgen we, indien de gebruiker bijvoorbeeld de tekst “Sean Connery” aanklikt, het volgende resultaat:

$$\text{alternatives}_{\text{filmdef}} \text{filmval}(4, 1, 0) = (\text{“Add acteur”}, \text{“Delete acteur”}, \text{“Edit String”})$$

Als de gebruiker de tekst “Actie” aanklikt, vinden we na uitwerking:

$$\begin{aligned} \text{alternatives}_{\text{filmdef}} \text{filmval}(0, 0) = & (\text{“Change To Actie”}, \text{“Change To Drama”}, \\ & \text{“Change To Komedie”}, \text{“Change To Thriller”}) \end{aligned}$$

De verschillende mogelijke wijzigingen kunnen nu in een menu getoond worden, en de gebruiker kan een bepaald alternatief selecteren. Op dat moment moet de documentwaarde aangepast worden. Hiervoor gebruiken we de functie `update`, die vier argumenten heeft. Het eerste argument is een rij typedefinities die het datatype definieert en het tweede argument is de huidige waarde van dat datatype. De overige argumenten duiden aan op welke locatie in het document de wijziging moet uitgevoerd worden en welke wijziging precies door de gebruiker geselecteerd werd. Het resultaat van de functie is een nieuwe waarde van hetzelfde datatype als het tweede argument. Merk op dat bij het wijzigen van karakterdata het laatste argument de waarde “Edit String” zal aannemen en dus niet voldoende informatie bevat om de nieuwe waarde van het datatype te construeren. De mogelijke alternatieven voor het wijzigen van karakterdata kunnen uiteraard niet eenvoudigweg in een menu weergegeven worden. Daarom zullen we voor het wijzigen van karakterdata een aparte teksteditor inbouwen. De functionaliteit van deze **subeditor** is beperkt en kan volledig gescheiden worden van de structurele editor die inwerkt op het document zelf. De procedure voor het wijzigen van een tekststring verloopt dan als volgt: de gebruiker selecteert de te wijzigen karakterdata door het aanklikken van de tekststring. Deze tekst wordt getoond in de subeditor, en kan daarin rechtstreeks gemanipuleerd worden. Vervolgens selecteert de gebruiker in het menu de optie “Edit String” en wordt de oorspronkelijke karakterdata vervangen door de nieuwe inhoud van de subeditor. Deze nieuwe karakterdata zou als een extra argument aan de functie `update` kunnen worden meegegeven, maar dit argument heeft dan slechts een zinvolle betekenis in een beperkt aantal gevallen. Een eenvoudigere oplossing bestaat er uit om het vierde argument hiervoor te gebruiken. Dit argument stelt normaal gezien het door de gebruiker

geselecteerde alternatief voor, maar in het geval van karakterdata is het enige alternatief “Edit String”, wat dus toch op zichzelf geen verdere informatie bevat. De functie `update` kunnen we nu als volgt definiëren:

$$\begin{aligned} \text{def update_} : \text{Typedef}^+ \ni t \rightarrow \text{toType } t \rightarrow \mathbb{N}^* \rightarrow \mathbb{A}^+ \rightarrow \text{toType } t \\ \text{with update}_t d p u = \text{updTypedef } t 0 d p u \end{aligned} \quad (2.28)$$

De hulpfunctie `updTypedef` genereert een waarde die identiek is aan de oorspronkelijke waarde, met uitzondering van één bepaald attribuut of constructor, waarvoor de functie `updStruct` wordt opgeroepen:

$$\begin{aligned} \text{def updTypedef :} \\ \text{Typedef}^+ \ni t \rightarrow \mathcal{D} t \ni i \rightarrow \text{typedefToType } t i \rightarrow \mathbb{N}^* \rightarrow \mathbb{A}^+ \rightarrow \text{typedefToType } t i \\ \text{with updTypedef } t i d p u = (p = \varepsilon) ? \phi \dagger \psi (p 0) \text{ where} \\ \phi = n \mapsto (t i n) \cup a \mapsto d a \cup c \mapsto k \mapsto (\text{newStruct } t \circ t i c k) \\ \text{where } k = (t i c)^-(\text{getNames } ^-(\text{drop } (\# \text{“Change_To_”}) u)) \\ \psi m = n \mapsto (t i n) \\ \cup a \mapsto (d a \otimes (m < \#(t i a) ? m \mapsto \text{updStruct } t (t i a m) (d a m) (\sigma p) u \dagger \varepsilon)) \\ \cup c \mapsto j \mapsto (d c j \otimes (m \geq \#(t i a) ? n \mapsto \text{updStruct } t (t i c j n) (d c j n) (\sigma p) u \dagger \varepsilon)) \text{ where} \\ j = \iota^-(\mathcal{D}(d c)) \\ n = m - \#(t i a) \end{aligned} \quad (2.29)$$

De hulpfuncties `drop` en `take` zijn als volgt gedefinieerd:

$$\begin{aligned} \text{drop} : \mathbb{N} \rightarrow A^* \rightarrow A^* & \qquad \text{take} : \mathbb{N} \rightarrow A^* \rightarrow A^* \\ \text{drop } 0 x = x & \qquad \text{take } 0 x = \varepsilon \\ \text{drop } n \varepsilon = \varepsilon & \qquad \text{take } n \varepsilon = \varepsilon \\ \text{drop } n (a \succ x) = \text{drop } (n - 1) x & \qquad \text{take } n (a \succ x) = a \succ \text{take } (n - 1) x \end{aligned}$$

De structuur van `updTypedef` weerspiegelt de structuur van `altTypedef`. Merk op dat de naam — aangeduid door het label `n` — voor een gegeven element vastligt en dus nooit kan gewijzigd worden. Bepaalde elementen kunnen uiteraard wel vervangen worden door elementen met een andere naam.

De functie `updStruct` heeft als argument een waarde van het type `StructType` en genereert de eigenlijke nieuwe waarde voor een element in de hiërarchische boomstructuur van het document.

De positiecode en het gekozen alternatief worden beiden gebruikt om zowel de aard als de locatie van de wijziging te bepalen:

$$\begin{aligned}
& \mathbf{def} \text{ updStruct} : \text{Typedef}^+ \ni t \rightarrow \text{StructType} \ni s \rightarrow \text{structToType } t s \rightarrow \mathbb{N}^* \rightarrow \mathbb{A}^+ \rightarrow \text{structToType } t s \\
& \mathbf{with} \text{ updStruct } t s d p u = \phi s \mathbf{where} \\
& \phi = (s : | (enum \mapsto (\mathbb{A}^+)^+) . \text{drop} (\# \text{“Change_To_”}) u) \\
& \quad \cup (s : | (default \mapsto \text{Record } (type \mapsto \text{StructType}, value \mapsto \mathbb{A}^*)) . \text{updStruct } t (s \text{ default type}) d p) \\
& \quad \cup (s : | (maybe \mapsto \text{StructType}) . u = (\text{“Add_”} ++ \text{getName } (s \text{ maybe})) ? \text{newStruct } (s \text{ maybe}) \\
& \quad \quad \dagger u = (\text{“Delete_”} ++ \text{getName } (s \text{ maybe})) ? \text{Nothing} \dagger \text{updStruct } t (s \text{ maybe}) d p) \\
& \quad \cup (s : | (list \mapsto \text{StructType}) . d = \varepsilon ? \tau (\text{newStruct } (s \text{ list})) \tag{a} \\
& \quad \quad \dagger ((\text{take } (p 0) (s \text{ list})) \tag{b} \\
& \quad \quad ++ (\quad u = (\text{“Add_”} ++ \text{getName } (s \text{ list})) ? (\text{newStruct } (s \text{ list}), d (p 0)) \tag{c1} \tag{c} \\
& \quad \quad \quad \dagger u = (\text{“Delete_”} ++ \text{getName } (s \text{ list})) ? \varepsilon \tag{c2} \\
& \quad \quad \quad \dagger \tau (\text{updStruct } t (s \text{ list}) (d (p 0)) (\sigma p)) \tag{c3} \\
& \quad \quad ++ (\text{drop } (1 + p 0) (s \text{ list})) \tag{d} \\
& \quad \cup (s : | (tuple \mapsto \text{StructType}^*) . d \otimes (p 0 \mapsto \text{updStruct } t (s \text{ tuple } (p 0)) (d (p 0)) (\sigma p))) \\
& \quad \cup (s : | (oneof \mapsto \text{StructType}^+) . u \in \mathcal{R} (\text{“Change To ”} \overleftarrow{++} (\text{getName} \circ s \text{ oneof})) ? \\
& \quad \quad \text{newStruct } (s \text{ oneof } ((\text{getName} \circ s \text{ oneof})^- (\sigma \# \text{“Change_To_”} u))) \\
& \quad \quad \dagger \text{updStruct } t (s \text{ oneof } j) (d j) p \mathbf{where} j = \iota^- (\mathcal{D} d) \\
& \quad \cup (s : | (string \mapsto \iota \text{ string}) . u \\
& \quad \cup (s : | (defined \mapsto \mathbb{A}^+) . \text{updTypedef } t ((t^\top \mathbf{n})^- (s \text{ defined})) d p \tag{2.30}
\end{aligned}$$

We bespreken opnieuw het label *list* ter illustratie. Als de lijst leeg was bij het selecteren van de wijziging, dan was het enige alternatief het toevoegen van een nieuw element. In dat geval wordt dus een nieuwe lijst aangemaakt met als enig element de verstekwaarde van het onderliggende type (a). Indien de lijst niet leeg was, dan duidt *p 0* op de index van het geselecteerde element uit de lijst. We voegen dan alles wat voor en na dit element komt opnieuw toe aan de gewijzigde lijst (regels b en d). Wat daar tussen komt hangt af van de gekozen wijziging (c). Indien gekozen werd om een element toe te voegen, voegen we een nieuw element en het geselecteerde element opnieuw toe (c1). Indien gekozen werd om een element te verwijderen, wordt het geselecteerde element niet opnieuw toegevoegd (c2). Als nog een andere keuze gemaakt werd, dan heeft die betrekking op het geselecteerde element. Dit element wordt dan verder gewijzigd en nadien

opnieuw aan de lijst toegevoegd (*c3*).

De functie `update` kan voor het filmdocument bijvoorbeeld als volgt opgeroepen worden:

$$\text{update}_{\text{filmdef}} \text{filmval} (0, 0) \text{ "Change To Drama"}$$

Het resultaat van deze oproep is een nieuwe waarde van het type *filmtype*, identiek aan *filmval*, met uitzondering van het gewijzigde genre.

De functies `new`, `showXml`, `alternatives` en `update` vormen de kernfunctionaliteit van de structurele editor. We hebben stap voor stap aangetoond hoe de informele karakterisatie van deze functionaliteit kan omgezet worden naar een reeks formele definities. Later zal blijken dat de overstap van deze Funmath-definities naar een concrete implementatie in Haskell vrij voor de hand liggend is. Naast de DTD-afhankelijke kernfunctionaliteit bevat de editor ook nog enkele meer algemene functies. Een typisch voorbeeld hiervan is de **undo**-functie. Tijdens het manipuleren van een document kan de gebruiker vergissingen maken en het is dus wenselijk dat de mogelijkheid bestaat om bepaalde wijzigingen ongedaan te maken. Om deze functie te modelleren, kunnen we ervan uitgaan dat we beschikken over een eindige buffer met lengte n , waarin we documentwaarden kunnen opslaan die recent door de gebruiker gewijzigd werden. Naast deze undo-buffer beschikken we ook over een even grote redo-buffer, zodat de gebruiker in beide richtingen kan navigeren tussen een beperkt aantal recente documentwaarden. De semantiek van een update-functie moet dan aangepast worden om met deze buffers rekening te houden. Als de documentwaarden het datatype A hebben, dan hebben de buffers u en r het type $\bigcup i : \square(n + 1). A^i$. Een *update*-functie die de huidige waarde *old* afbeeldt op de nieuwe waarde *new* kan dan als volgt gedefinieerd worden:

$$\begin{aligned} \mathbf{def} \text{ update} : A \times A^* \times A^* &\rightarrow A \times A^+ \times A^* \\ \mathbf{with} \text{ update} (old, u, r) &= (new, \text{take } n (old \succ u), \varepsilon) \end{aligned}$$

De nieuwe undo-buffer ontstaat door de huidige documentwaarde vooraan aan de undo-buffer toe te voegen en van het resultaat enkel de eerste n waarden over te houden. De redo-buffer wordt geleegd omdat, bij het uitvoeren van een nieuwe wijziging, de eerder ongedaan gemaakte wijzigingen niet langer te herstellen zijn. De *undo*-functie ziet er zelf als volgt uit:

$$\begin{aligned} \mathbf{def} \text{ undo} : A \times A^+ \times A^* &\rightarrow A \times A^* \times A^+ \\ \mathbf{with} \text{ undo} (old, u, r) &= (u \ 0, \sigma u, old \succ r) \end{aligned}$$

De huidige documentwaarde wordt door deze functie opgeslagen in de redo-buffer en vervangen door de laatst in de undo-buffer opgeslagen documentwaarde, die uit de undo-buffer verwijderd wordt. De *redo*-functie maakt het mogelijk de *undo*-bewerkingen zelf ongedaan te maken:

def *redo* : $A \times A^* \times A^+ \rightarrow A \times A^+ \times A^*$
with *redo* (*old*, *u*, *r*) = (*r* 0, *old* \succ *u*, σ *r*)

De huidige documentwaarde wordt dan door de *redo*-functie opgeslagen in de undo-buffer en vervangen door de laatst in de redo-buffer opgeslagen documentwaarde. Deze formele specificaties bieden niet alleen een handige referentie voor een concrete implementatie, maar kunnen ook gebruikt worden om bepaalde eigenschappen formeel aan te tonen. Zo verwachten we dat voor geldige argumenten *old*, *u* en *r* onder meer de volgende gelijkheden gelden.

$$\begin{aligned} (\text{undo} \circ \text{update}) (\text{old}, u, r) 0 &= \text{old} \\ (\text{redo} \circ \text{undo}) (\text{old}, u, r) &= (\text{old}, u, r) \\ (\text{undo} \circ \text{redo}) (\text{old}, u, r) &= (\text{old}, u, r) \end{aligned}$$

We bewijzen als voorbeeld de tweede gelijkheid:

$$\begin{aligned} (\text{redo} \circ \text{undo}) (\text{old}, u, r) &= \langle \text{definitie } \circ \rangle && \text{redo} (\text{undo} (\text{old}, u, r)) \\ &= \langle \text{definitie } \text{undo} \rangle && \text{redo} (u 0, \sigma u, \text{old} \succ r) \\ &= \langle \text{definitie } \text{redo} \rangle && ((\text{old} \succ r) 0, u 0 \succ \sigma u, \sigma (\text{old} \succ r)) \\ &= \langle (a \succ x) 0 = a \rangle && (\text{old}, u 0 \succ \sigma u, \sigma (\text{old} \succ r)) \\ &= \langle \sigma (a \succ x) = x \rangle && (\text{old}, u 0 \succ \sigma u, r) \\ &= \langle x = (x 0 \succ \sigma x) \rangle && (\text{old}, u, r) \end{aligned}$$

Door het gebruik van bovenstaande methode kunnen ontwerpfouten reeds in een vroeg stadium opgemerkt worden en implementatiefouten dus vermeden worden.

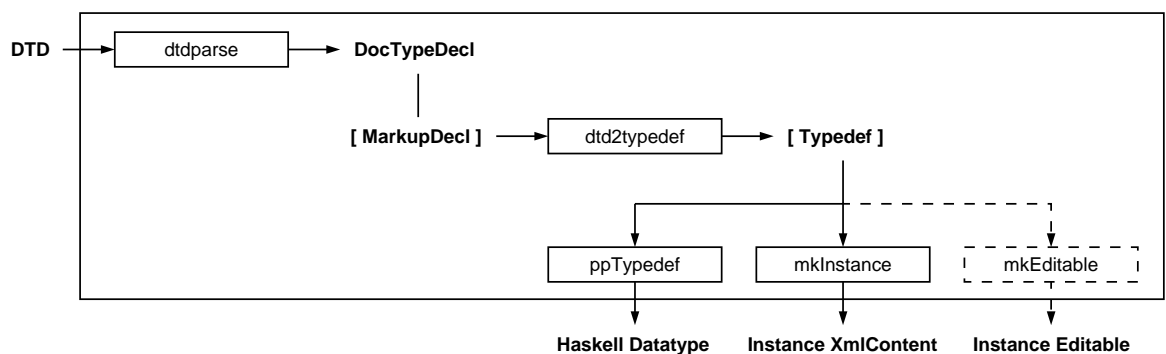
Hoofdstuk 3

Implementatie in Haskell

3.1 Implementatie van de editor-functionaliteit

Voor de concrete implementatie van de editor in Haskell, baseren we ons op de formele beschrijving van de functionaliteit uit hoofdstuk 2. We zullen zowel de gelijkenissen als de verschillen tussen de formele beschrijving en de implementatie toelichten en verantwoorden.

Om het programmeerwerk te beperken, hergebruiken we de code uit de HaXml-toolkit en voegen we hieraan de nodige functies toe. Specifiek breiden we het programma `DtdToHaskell` uit met nieuwe functionaliteit. Dit programma genereert Haskell broncode op basis van een gegeven DTD. De broncode omvat de definities van de datatypes die met de DTD kunnen geassocieerd worden, aangevuld met de declaraties die nodig zijn opdat de datatypes tot de klassen `XmlContent` en `Editable` zouden behoren. De interne werking van het programma staat afgebeeld in Fig. 3.1. De nieuwe functionaliteit zit vervat in de functie `mkEditable`.



Figuur 3.1: Opbouw van het programma `DtdToHaskell`

Een DTD wordt door de functie `dtddata` omgezet naar een Haskell-representatie onder de vorm van een waarde van het type `DocTypeDecl`. Deze waarde bevat onder meer een lijst instanties van het type `MarkupDecl`, die de verschillende specificatieregels uit de DTD voorstellen. Deze instanties worden door de functie `dtddata2typedef` gemapt op waarden van het type `Typedef`. Bij de formele karakterisatie hebben we ons voor de definitie van de types `Typedef` en `StructType` gebaseerd op de gelijknamige Haskell-types, die er als volgt uitzien:

```
data Typedef = DataDef Bool Name AttrFields Constructors
             | EnumDef Name [Name]
type Constructors = [(Name, [StructType])]
type AttrFields  = [(Name, StructType)]
data StructType = Maybe StructType
                | Defaultable StructType String
                | List StructType
                | Tuple [StructType]
                | OneOf [StructType]
                | String
                | Defined Name
data Name = Name { xName :: String, hName :: String }
```

De enige wezenlijke verschillen met uitdrukkingen (2.2) en (2.3) zijn de aparte behandeling van enumeraties — omwille van efficiëntieredenen — en de extra voorkomens van het `Name`-type — om gemakkelijk de onderliggende elementen te kunnen identificeren. Een naam bestaat uit een XML-naam, zoals gespecificeerd in de DTD, en een daarvan afgeleide Haskell-naam, zoals gebruikt in de datatypes. Bij de modellering in `Funmath` wordt een dergelijk onderscheid niet gemaakt omdat we daar niet gebonden zijn aan bepaalde naamgevingsconventies zoals in Haskell wel het geval is.

De functie `ppTypedef` (“pretty-print typedef”) werkt in op de lijst van het type `[Typedef]` en genereert op basis hiervan de broncode voor een Haskell datatype. Zo wordt voor het filmdocument het volgende datatype gegenereerd:

```
data Film = Film Film_Attrs Titel Regisseur (Maybe Jaar) [Acteur]
data Film_Attrs = Film_Attrs { filmGenre :: Film_Genre }
data Film_Genre = Film_Genre_Actie | Film_Genre_Thriller |
```

```

                                Film_Genre_Komedie | Film_Genre_Drama
newtype Titel = Titel String
newtype Regisseur = Regisseur String
newtype Jaar = Jaar String
newtype Acteur = Acteur String

```

Een declaratie van de vorm `data D = D1 A B | D2 B | D3` bepaalt in Haskell een somdatatype met drie alternatieven, waarbij elk alternatief een producttype voorstelt. Een producttype bestaat uit een dataconstructor (`D1`, `D2`, `D3`) en eventueel subtypes (`A`, `B`). De naam van elke constructor moet uniek zijn, maar er wordt niet geëist dat deze naam verschilt van de naam van het datatype. Er is geen verwarring mogelijk omdat typeconstructoren en dataconstructoren in verschillende naamruimten zitten. `data D = D A B` is dus een geldige declaratie van een datatype. Het verschil tussen `data` en `newtype` is subtiel en heeft te maken met efficiëntieredenen.

Het bovenstaande datatype is vergelijkbaar met het datatype *filmtype* (2.12) uit de formele beschrijving. In Haskell merken we dat het type `Film` is samengesteld uit een aantal subtypes, die elk afzonderlijk overeenkomen met een specificatieregule uit de DTD.

Door de functie `mkInstance` worden Haskell-declaraties gegenereerd met als resultaat dat het datatype tot de klasse `XmlContent` behoort:

```

class XmlContent a where
    toElem    :: a -> [Content]
    fromElem  :: [Content] -> (Maybe a, [Content])

```

Voor elk gegenereerd datatype `a` worden de functies `toElem` en `fromElem` gedefinieerd, die gebruikt worden in de implementatie van de functies `readXml` en `writeXml` voor het inlezen en uitschrijven van waarden van het datatype:

```

readXml  :: (XmlContent a) => FilePath -> IO a
writeXml :: (XmlContent a) => FilePath -> a -> IO ()

```

De functies `readXml` en `writeXml` worden in de editor gebruikt bij het openen en bewaren van XML-documenten en vormen dus een implementatie van de formeel gedefinieerde `open`- en `save`-functies.

De nieuwe functionaliteit zit vervat in de functie `mkEditable`, die de nodige code genereert zodat het nieuwe datatype tot de klasse `Editable` behoort:

```

class Editable a where
  new :: a
  con :: [Int] -> [Int] -> Int -> a -> [( [Int], Int, String, ContentType)]
  alt :: [Int] -> a -> [String]
  upd :: [Int] -> String -> a -> a

```

De functies die door de klasse `Editable` gedefinieerd worden kunnen dan vrijwel rechtstreeks gebruikt worden als implementatie van de kernfunctionaliteit van de editor:

```

new          :: (Editable a) => a
showXml     :: (Editable a) => a -> [ContentElement]
alternatives :: (Editable a) => a -> [Int] -> [String]
update      :: (Editable a) => a -> [Int] -> String -> a

```

Merk op dat deze Haskell-functies qua typering volledig overeenkomen met de gelijknamige formeel gedefinieerde functies, op één uitzondering na: de lijst van het type `[Typedef]` komt niet expliciet voor in bovenstaande functies. De verklaring hiervoor is dat omwille van efficiëntie-redenen voor elke DTD een nieuwe verzameling functies gegenereerd wordt — gebaseerd op de verschillende specificatieregels uit de DTD — en dat de resulterende functies zelf dus niet langer DTD-afhankelijk zijn. Indien we bijvoorbeeld de formele definitie van `update` rechtstreeks zouden implementeren als een DTD-afhankelijke functie, dan zou dit betekenen dat bij elke wijziging aan een document de DTD geëvalueerd wordt. Het volstaat echter de DTD slechts eenmaal te evalueren — bij het uitvoeren van de functie `mkEditable` — en op basis van deze evaluatie een `update`-functie te genereren die zelf niet langer DTD-afhankelijk is. Conceptueel kunnen we stellen dat bovenstaande functies partiële toepassingen zijn van de DTD-afhankelijke varianten. De eigenlijke implementatie van de functies vertoont eveneens sterke gelijkenissen met de formele specificaties. Er zijn echter ook enkele fundamentele verschillen, die vooral te maken hebben met technische aspecten van Haskell. We zullen dit illustreren aan de hand van de implementatie van de formeel gedefinieerde functie `toType`.

De functie `toType` neemt als argument een rij van het type `Typedef+` en geeft als resultaat een datatype. Hierbij wordt gebruik gemaakt van een hulpfunctie `typedefToType`, die een datatype associeert met een gegeven typedefinitie. Vooreerst wordt deze hulpfunctie toegepast op de typedefinitie die het documentelement karakteriseert. Omdat de functie `typedefToType` de volledige

rij typedefinities als extra argument heeft, kan de functie vervolgens waar nodig recursief toegepast worden voor andere typedefinities — in praktijk de typedefinities corresponderend met de onderliggende elementen.

In Haskell wordt de functie `toType` geïmplementeerd door `(map ppTypedef)`, toegepast op een lijst van het type `[Typedef]` (zie Fig. 3.1). Het resultaat hiervan vormt de broncode van een Haskell datatype. De functie `ppTypedef` zelf kunnen we vergelijken met `typedefToType`. Een belangrijk verschil is dat `ppTypedef` rechtstreeks (niet recursief) voor elke typedefinitie een afzonderlijk datatype genereert. Deze datatypes zijn uiteraard onderling sterk verbonden maar kunnen wel onafhankelijk van elkaar gegenereerd worden. De functie `ppTypedef` neemt dus niet de volledige lijst typedefinities als extra argument, het volstaat de naam van de onmiddellijke opvolgers in de elementenboom te kennen, en deze informatie zit vervat in elke afzonderlijke typedefinitie.

Ter illustratie beschouwen we de film-DTD. In hoofdstuk 2 wordt uiteengezet hoe het datatype *filmtype* recursief opgebouwd wordt door vooreerst de functie `typedefToType` toe te passen op de typedefinitie geassocieerd met het film-element. In uitdrukking (2.11) is te zien hoe als gevolg hiervan vervolgens de functie `typedefToType` onder meer recursief toegepast wordt op de typedefinitie geassocieerd met het titel-element (*filmdef 2*). Het bijhorende Haskell-datatype ontstaat echter door onafhankelijk voor elke typedefinitie een apart subtype te genereren. Zo wordt voor de typedefinitie geassocieerd met het titel-element het type `Titel` gedefinieerd. Uit de typedefinitie geassocieerd met het film-element kunnen we afleiden dat het titel-element een onderliggend element vormt van het film-element, zodat we bij het definiëren van het type `Film` weten dat `Titel` een subtype is van `Film`. De types `Film` en `Titel` kunnen echter onafhankelijk gegenereerd worden.

Aangezien datatypes in Haskell niet dynamisch kunnen aangemaakt worden, geeft de functie `ppTypedef` als resultaat de Haskell broncode voor het gewenste datatype. Door deze broncode te includeren kan het datatype vervolgens gebruikt worden in een Haskell-programma. Dit houdt onder meer in dat een DTD-afhankelijk programma voor elke nieuwe DTD opnieuw moet gelinkt worden. Vanwege deze onrechtstreekse aanpak zijn de functies van het programma `DtdToHaskell` over het algemeen minder doorzichtig dan de formele specificaties. Zo heeft de functie `toType` als resultaattype de verzameling \mathcal{T} van alle mogelijke datatypes, terwijl de functie `ppTypedef` als resultaattype `Doc` heeft, een abstracte voorstelling van een tekststring.

De functie `ppTypedef` maakt omwille van de efficiëntie een onderscheid tussen typedefinities met

of zonder attributen en tussen typedefinities met één of meerdere constructoren, maar in essentie wordt de hulpfunctie `ppST` gemapt op de verschillende attributen en constructoren. Deze aanpak is analoog met de beeldpuntdefinitie van de functie `typedefToType`, waarbij de functie `ppST` een implementatie vormt van de formeel gedefinieerde functie `structToType`:

```
ppST :: StructType -> Doc
ppST (Defaultable st _) = parens (text "Defaultable" <+> ppST st)
ppST (Maybe st)       = parens (text "Maybe" <+> ppST st)
ppST (List st)        = text "[" <> ppST st <> text "]"
ppST (Tuple sts)      = parens (commaList (map ppST sts))
ppST (OneOf sts)      = parens (text "OneOf" <> text (show (length sts)) <+>
                                hsep (map ppST sts))
ppST String           = text "String"
ppST (Defined n)      = ppHName n
```

Bij de definitie van `ppST` worden een aantal hulpfuncties gebruikt uit een bibliotheek voor het opmaken van tekst. Het type en de betekenis van deze functies staat weergegeven in Tab. 3.1. Merk op dat de semantiek van de functie `ppST` zoals verwacht sterke gelijkenissen vertoont met

<code>text</code>	<code>String -> Doc</code>	omzetting naar primitieve Doc-waarde
<code>parens</code>	<code>Doc -> Doc</code>	toevoegen van haakjes aan Doc-waarde
<code>commaList</code>	<code>[Doc] -> Doc</code>	scheiden van lijst Doc-waarden door komma's
<code>hsep</code>	<code>[Doc] -> Doc</code>	scheiden van lijst Doc-waarden door spaties
<code>(<>)</code>	<code>Doc -> Doc -> Doc</code>	samenvoegen van twee Doc-waarden zonder spatie
<code>(<+>)</code>	<code>Doc -> Doc -> Doc</code>	samenvoegen van twee Doc-waarden met spatie

Tabel 3.1: Enkele “Pretty Printer Combinators”

de beeldpuntdefinitie van de functie `structToType`. Uiteraard wordt hier wel Haskell gebruikt in plaats van Funmath. Het enige echt relevante verschil situeert zich op de laatste regel. In de functie `structToType` wordt voor het label *defined* een recursieve oproep van de functie `typedefToType` gebruikt om het datatype van het onderliggende element te construeren. In de Haskell-implementatie weten we echter dat de functie `ppTypedef` onafhankelijk zal opgeroepen worden voor alle typedefinities, zodat het volstaat om de Haskell-naam van het onderliggende element in te vullen.

In tabel 3.2 wordt een overzicht gegeven van het verband tussen de formele beschrijving in Funmath en de implementatie in Haskell. De functies `ppTypedef` en `mkInstance` behoren tot de HaXml-toolkit, de functie `mkEditable` werd door ons geïmplementeerd. Deze Haskell-functies vormen geen rechtstreekse implementatie van de Funmath-functies maar genereren Haskell-broncode. In deze broncode worden de partiële toepassingen gedefinieerd van de corresponderende DTD-afhankelijke Funmath-functies.

Funmath	Haskell	Gegeneerd door
<code>toType</code>	Haskell datatype	<code>map ppTypedef</code>
<code>open</code>	<code>readXml</code>	<code>map mkInstance</code>
<code>save</code>	<code>writeXml</code>	
<code>new</code>	<code>new</code>	<code>map mkEditable</code>
<code>showXml</code>	<code>showXml</code>	
<code>alternatives</code>	<code>alternatives</code>	
<code>update</code>	<code>update</code>	

Tabel 3.2: Verband tussen formele beschrijving en implementatie

Een overzicht van de verschillende hulpfuncties is te vinden in tabel 3.3.

Funmath	Haskell	Funmath	Haskell
<code>typedefToType</code>	<code>ppTypedef</code>	<code>structToType</code>	<code>ppST</code>
<code>newTypedef</code>	<code>mkEditable</code>	<code>newStruct</code>	<code>newST</code>
<code>showTypedef</code>		<code>showStruct</code>	<code>conST</code>
<code>altTypedef</code>		<code>altStruct</code>	<code>altST</code>
<code>updateTypedef</code>		<code>updStruct</code>	<code>updST</code>

Tabel 3.3: Verband tussen de verschillende hulpfuncties

3.2 Implementatie van de grafische gebruikersinterface

Eens we beschikken over een implementatie van de editor-functionaliteit, is er behoefte aan een interface die deze functionaliteit toegankelijk maakt voor de gebruiker. Omwille van de gebruiks-

vriendelijkheid opteren we hier voor een grafische gebruikersinterface (GUI). Haskell voorziet geen standaard mechanisme voor het ontwikkelen van GUI's — zoals bijvoorbeeld “Swing” voor Java, maar er bestaan wel verschillende externe toolkits. Voor de implementatie van de editor hebben we gekozen voor de **FranTk** toolkit [34], die voldoet aan de volgende vooropgestelde eisen:

- Platform-onafhankelijke code
- Declaratieve programmeerstijl
- Wiskundige fundering

FranTk steunt op Tcl-Tk, een platformonafhankelijke en robuuste taal voor het ontwikkelen van GUI's. Programma's die gebruik maken van FranTk kunnen zonder enige aanpassing gecompileerd worden op elk platform waarvoor Tcl-Tk beschikbaar is. De toolkit laat toe om op declaratieve wijze een GUI te implementeren in Haskell door het gebruik van de GUI-monade, een uitbreiding van de standaard IO-monade [26]. Waarden van het type `GUI a` stellen acties voor die een waarde teruggeven van het type `a` en een neveneffect kunnen hebben op de gebruikersinterface. Bij het ontwikkelen van een GUI met FranTk kan de programmeur gebruik maken van concepten uit de taal **Functional Reactive Animation** (FRAN) [14], zoals **listeners** en **events**, die toelaten een tijdsafhankelijk systeem te modelleren. Deze begrippen voldoen aan bepaalde algebraïsche wetten, waarmee men de correctheid van een GUI formeel kan aantonen. FranTk voorziet een implementatie van deze concepten, onder de vorm van elementaire bouwblokken, die door de programmeur kunnen gecombineerd worden tot krachtigere constructies.

3.2.1 Basisconcepten

Een waarde van het type `BVar a` is een abstracte voorstelling van een veranderlijke toestand van het type `a`. Zo gebruiken we in de editor bijvoorbeeld een waarde van het type `BVar String` om de veranderlijke toestand voor te stellen die de naam representeert van het geopende document. Bij de creatie van een `BVar` moet een initiële waarde voor de onderliggende toestand opgegeven worden:

```
mkBVar :: a -> GUI (BVar a)
```

Merk op dat de initialisatie van een `BVar` een neveneffect op de GUI veroorzaakt en dat het resultaat van de functie `mkBVar` dan ook het type `GUI (BVar a)` heeft in plaats van `BVar a`. Het concept van veranderlijke toestand kan namelijk niet buiten de GUI-monade (of de IO-monade) gemodelleerd worden, omdat daar de neveneffecten die de toestand wijzigen niet mogelijk zijn.

Met elke `BVar` kunnen we een signaal, een event en een listener associëren:

```
bvarBehavior :: BVar a -> Behavior a
bvarEvent    :: BVar a -> Event a
bvarInput    :: BVar a -> Listener a
```

Een signaal van het type `Behavior a` is conceptueel een continue tijdsafhankelijke waarde van het type `a`. We kunnen het type van een dergelijk signaal vereenvoudigd voorstellen als `Time -> a`. Deze continue representatie is minder geschikt voor een waarde die op discrete tijdstippen verandert. Een discrete tijdsafhankelijke waarde stellen we daarom voor door een event, conceptueel een stroom of een lijst tijd-waardeparen van het type `[(Time,a)]`. Een event dat gekoppeld is aan een `BVar` genereert een nieuw tijd-waardepaar wanneer de waarde van de onderliggende `BVar` verandert. Listeners kunnen gebruikt worden om de interactie met de omgeving te realiseren. Een waarde van het type `Listener a` consumeert waarden van het type `a` met een IO-actie als neveneffect. In het geval van een `BVar` zal het neveneffect eruit bestaan dat de waarde van de onderliggende `BVar` gewijzigd wordt naar de geconsumeerde waarde. We kunnen het type `Listener a` conceptueel voorstellen als `a -> IO ()`, hoewel het in werkelijkheid om een abstract type gaat. Dit betekent dat de constructoren van het datatype niet rechtstreeks beschikbaar zijn voor de gebruiker. Een uitgebreide formele beschrijving van deze verschillende concepten wordt gegeven in [14].

Een `BVar` is een instantie van de klasse `Has_Event`:

```
class Has_Event c where
  input :: c a -> Listener a
  event :: c a -> Event a
```

We kunnen dus kortweg `input` en `event` gebruiken om respectievelijk de listener en het event van een `BVar` op te vragen, in plaats van de meer expliciete functies `bvarInput` en `bvarEvent`.

In wat volgt bespreken we enkele veel gebruikte operatoren voor listeners en events.

3.2.2 De listener-operatoren

De meest eenvoudige listener is `neverL`, die waarden consumeert zonder daarbij enig neveneffect te veroorzaken:

```
neverL :: Listener a
```

Twee listeners `l1` en `l2` van hetzelfde type kunnen samengevoegd worden tot een nieuwe listener met behulp van `mergeL`:

```
mergeL :: Listener a -> Listener a -> Listener a
```

De resulterende listener consumeert waarden van het type `a` en geeft deze waarden door aan zowel `l1` als `l2`. Een voor de hand liggende uitbreiding is een listener die geconsumeerde waarden doorgeeft aan een lijst listeners, wat gerealiseerd kan worden door de operator `anyL`:

```
anyL :: [Listener a] -> Listener a  
anyL = foldr mergeL neverL
```

Een volgende belangrijke operator is `mapL`, die een listener van het type `Listener b` omzet naar een listener van het type `Listener a` door het toepassen van een gegeven functie op de geconsumeerde waarden:

```
mapL :: (a -> b) -> Listener b -> Listener a
```

Het resultaat van `mapL f lb` is een listener die waarden van het type `a` consumeert, op deze waarden de functie `f` uitvoert, en de resulterende waarden doorgeeft aan `lb`. Merk op dat de volgorde van de argumenten voor operatoren uit de listener algebra tegengesteld is aan de volgorde waarin de waarden geconsumeerd worden.

De operator `tellL` is een speciale versie van `mapL`, die gebruikt wordt om een constante waarde door te geven aan een listener:

```
tellL :: Listener b -> b -> Listener a  
tellL lb a = mapL (const a) lb
```

De functie `filterL` kan gebruikt worden om enkel die geconsumeerde waarden door te geven die voldoen aan een gegeven predikaat:

```
filterL :: (a -> Bool) -> Listener a -> Listener a
```

Met de functie `fromListL` kunnen we een listener die waarden van het type `a` consumeert omzetten naar een listener die lijsten van het type `[a]` consumeert en elk element afzonderlijk doorgeeft naar de gegeven listener:

```
fromListL :: Listener a -> Listener [a]
```

Wanneer we bij het consumeren van een waarde een expliciete IO-actie willen uitvoeren, kunnen we gebruik maken van de operator `mkL`:

```
mkL :: (a -> IO ()) -> Listener a
```

Ten slotte geven we nog een operator waarvan het resultaat een listener is die bij het consumeren van een waarde de huidige waarde van een signaal evalueert en mee doorgeeft aan een gegeven listener:

```
snapshotL :: Behavior a -> Listener (a,b) -> Listener b
```

3.2.3 De event-operatoren

Voor de meeste operatoren uit de listener algebra, bestaat een analoge operator in de event algebra. De volgorde van de argumenten komt nu wel overeen met de volgorde waarin de waarden gegenereerd worden.

Een event dat geen tijd-waardeparen genereert is `neverE`:

```
neverE :: Event a
```

Twee events `e1` en `e2` van hetzelfde type kunnen samengevoegd worden tot een nieuw event door de operator `mergeE`:

```
mergeE :: Event a -> Event a -> Event a
```

Wanneer `e1` of `e2` een tijd-waardepaar genereert, dan zal het nieuwe event hetzelfde tijd-waardepaar genereren. Opnieuw kunnen we dit principe uitbreiden naar een lijst events:

```
anyE :: [Event a] -> Event a
```

De operator `mapE` zet een event van het type `Event a` om naar een event van het type `Event b` door op de gegenereerde waarden eerst een functie toe te passen:

```
mapE :: (a -> b) -> Event a -> Event b
```

Wanneer het event `ea` een tijd-waardepaar (t, a) genereert, zal het event `mapE f ea` het tijd-waardepaar $(t, f a)$ genereren.

De operator `tellE` is vergelijkbaar met `mapE`, maar negeert de gegenereerde waarde en vervangt ze door een gegeven vaste waarde:

```
tellE :: Event a -> b -> Event b  
tellE ea b = mapE (const b) ea
```

Wanneer het event `ea` een tijd-waardepaar (t, a) genereert, zal het event `tellE ea b` het tijd-waardepaar (t, b) genereren met vaste waarde `b`.

Met de functie `fromListE` kunnen we een event dat lijsten van het type `[a]` genereert omzetten naar een event van het type `Event a` dat voor elk element uit de lijst een afzonderlijk tijd-waardepaar genereert:

```
fromListE :: Event [a] -> Event a
```

De operator `snapshotE` converteert een event van het type `Event a` naar een event van het type `Event (a,b)`. De gegenereerde waarde bestaat dan enerzijds uit de waarde gegenereerd door het oorspronkelijke event (type `a`) en anderzijds uit de huidige waarde van een signaal (type `b`):

```
snapshotE :: Event a -> Behavior b -> Event (a,b)
```

In praktijk zullen listeners en events zelden geïsoleerd gebruikt worden en is er steeds een vorm van onderlinge interactie. Zo kan een listener de waarden consumeren die gegenereerd worden door een event of omgekeerd. Zoals reeds gezegd kan een `BVar` gebruikt worden om listeners en events aan elkaar te koppelen. In dat geval is er sprake van een onderliggende veranderlijke toestand waarmee een listener, een event en een signaal kunnen geassocieerd worden.

3.2.4 De editor GUI

FranTk voorziet een aanzienlijke collectie grafische componenten, zoals knoppen, menu's en vensters, grotendeels gebaseerd op Tcl. Een dergelijke component noemt men een `Widget`. Elke `Widget` wordt aangemaakt door een actie die een neveneffect heeft op de GUI, in de terminologie van FranTk een `Component` genoemd:

```
type Component = GUI Widget
```

Een bijzondere `Widget` is een `WindowWidget` die een venster voorstelt. Een `WindowWidget` wordt gecreëerd door een `WComponent` en kan op het scherm weergegeven worden door de functie `render`:

```
type WComponent = GUI WindowWidget
render :: WComponent -> GUI ()
```

De actie die de GUI realiseert kunnen we omzetten naar een standaard IO-actie door de functie `start`. Omdat de functies `render` en `start` vaak samen gebruikt worden, is er ook een samengestelde functie `display`:

```
start :: GUI () -> IO ()
display :: WComponent -> IO ()
display = start . render
```

De implementatie van de editor ziet er als volgt uit:

```
main :: IO ()
main = display editor
```

De waarde `editor` heeft het vereiste type `WComponent` en bestaat uit een opeenvolging van GUI-acties, aaneengeschakeld door de `do`-syntax uit de IO-monade. In een eerste fase worden de nodige `BVars` geïnitieerd:

```
editor :: WComponent
editor = do doc <- mkBVar (new::Document)
          undo <- mkBVar ([]::[Document])
```

```
redo <- mkBVar ([]::[Document])
wTitle <- mkBVar "New File"
saved <- mkBVar False
```

Het type `Document` wordt bij het genereren van de datatypes voor een gegeven DTD gelijkgesteld aan het datatype van het documentelement, bijvoorbeeld `type Document = Film`. De veranderlijke toestand van het document dat gewijzigd wordt in de editor wordt voorgesteld door de waarde `doc` van het type `BVar Document`. Deze `BVar` wordt geïnitieerd met de verstekwaarde voor het type `Document`, gegeven door de functie `new`. De waarden `undo` en `redo` hebben het type `BVar [Document]` en stellen de respectievelijke toestand van de undo- en de redo-buffers voor. Beide buffers zijn initieel leeg. De `BVar wTitle` representeert de veranderlijke toestand van de naam van het geopende document en heeft het type `BVar String`. De `BVar saved` ten slotte heeft het type `BVar Bool` en wordt gebruikt om bij te houden of een nieuw document reeds werd opgeslagen.

Na de creatie van de `BVars` worden listeners gedefinieerd die zullen gebruikt worden om een aantal basiscommando's in de editor te realiseren. Deze listeners hebben allen het type `Listener ()`, wat inhoudt dat ze de waarde `()` van het lege type `()` consumeren. Dergelijke listeners kunnen gekoppeld worden aan een knop, die zich bijvoorbeeld in een selectiemenu bevindt. Telkens wanneer de knop wordt aangeklikt, zal de bijhorende listener de waarde `()` te verwerken krijgen. De listener `newL` wordt gebruikt bij het openen van nieuw document en zal aan de verschillende `BVars` opnieuw de initiële waarde toekennen:

```
let newL = anyL [tellL (input doc) (new::Document),
               tellL (input undo) [],
               tellL (input redo) [],
               tellL (input wTitle) "New File",
               tellL (input saved) False]
```

We kunnen de werking van bovenstaande listener op twee manieren interpreteren: als we van binnenuit vertrekken, merken we dat `(input saved)` een listener is van het type `Listener Bool`. Deze listener wordt door de functie `tellL` omgezet naar een listener van het type `Listener a`, met `a` een willekeurig type. Op analoge manier worden de listeners geassocieerd met de overige `BVars` elk omgezet naar een listener van het type `Listener a`. De bekomen lijst van het type

[`Listener a`] wordt ten slotte omgezet naar een listener van het type `Listener a` door de functie `anyL`. Zodra de bekomen listener gekoppeld wordt aan een knop (zie verder), zal het type `a` vastgelegd worden op `()`. Als we van buitenaf vertrekken, kunnen we nagaan wat er gebeurt met een waarde die door `newL` geconsumeerd wordt. Deze waarde wordt door `anyL` doorgegeven aan elke listener uit de lijst, waaronder ook `tellL (input saved) False`. Deze laatste listener zal de geconsumeerde waarde negeren en de vaste waarde `False` verder doorgeven aan de listener `(input saved)`, die de toestand van de onderliggende `BVar saved` zal aanpassen. De volgende listener die we bespreken is `saveL`, waarvan de werking afhankelijk is van de huidige toestand van de `BVar saved`. Om dit te realiseren gebruiken we de hulpfunctie `switchL`, met als argumenten een `BVar` van het type `BVar Bool` en twee listeners. Het resultaat van deze functie is een nieuwe listener die bij het consumeren van een waarde de huidige toestand van de gegeven `BVar` zal evalueren en op basis daarvan de geconsumeerde waarde zal doorgeven aan één van de twee gegeven listeners:

```
switchL :: BVar Bool -> Listener a -> Listener a -> Listener a
```

De implementatie van `saveL` ziet er dan als volgt uit:

```
let saveL = switchL saved
    (snapshotL (bvarBehavior wTitle)
     (snapshotL (bvarBehavior doc)
                (mkL (\(s,d) -> writeXml s d))))
    saveAsL
```

Als het document reeds opgeslagen werd, dan moet enkel de nieuwe waarde bewaard worden. In dat geval wordt de toestand van de `BVars wTitle` en `doc` geëvalueerd. De resulterende waarden `s` en `d` hebben respectievelijk het type `String` en `Document` en kunnen dus rechtstreeks gebruikt worden als argumenten voor de functie `writeXml`. We herinneren eraan dat deze functie het type `(XmlContent a) => FilePath -> a -> IO ()` heeft, zodat we de functie `mkL` kunnen gebruiken om een listener te construeren.

Als het te bewaren document een nieuw document is dat nog niet eerder werd opgeslagen, dan moeten eerst een locatie en een naam aan de gebruiker gevraagd worden. In dat geval moet `saveL` zich gedragen als de listener `saveAsL`, die we hier niet in detail bespreken omdat het

bewaren van een nieuw document slechts weinig verschilt van het bewaren van een reeds vroeger opgeslagen bestand.

De listener `openL` wordt gebruikt bij het openen van een bestand. Deze listener vertoont sterke gelijkenissen met `newL` en `saveAsL` en zullen we dan ook niet verder bespreken. Om de editor af te sluiten gebruiken we `quitL`:

```
quitL <- mkGUIL (const quit)
```

De functie `mkGUIL` is een variant van `mkL` met als type $(a \rightarrow \text{GUI } ()) \rightarrow \text{GUI } (\text{Listener } a)$, die we hier moeten gebruiken omdat de functie `quit` het type `GUI ()` en niet `IO ()` heeft.

Ten slotte bespreken we de listener `undoL`:

```
let undoL = (snapshotL (bvarBehavior undo)
              (mapL snd (filterL (/= []) (anyL ulist)))) where
  ulist = [mapL head (input doc), mapL tail (input undo),
           snapshotL (bvarBehavior doc)
           (snapshotL (bvarBehavior redo)
                     (mapL (\((_,d),r)-> d:r) (input redo)))]
```

Door de functie `snapshotL (bvarBehavior undo)` wordt de huidige toestand van de undo-buffer geëvalueerd en toegevoegd aan de door `undoL` geconsumeerde waarde. De functie `mapL snd` heeft als effect dat enkel de undo-buffer verder wordt doorgegeven en de functie `filterL (/= [])` zorgt ervoor dat dit enkel gebeurt indien de buffer niet leeg is. Indien de undo-buffer leeg is kan geen zinvolle actie ondernomen worden. De functie `anyL` zal de niet-lege undo-buffer doorspelen aan elke listener uit `ulist`. Het doorgeven aan de eerste listener uit `ulist` heeft als resultaat dat het eerste element uit de geconsumeerde undo-buffer geselecteerd wordt door de functie `head` en doorgegeven wordt aan de listener `(input doc)`. Het gevolg hiervan is dat de toestand van het document gewijzigd wordt in de geconsumeerde waarde. De tweede listener uit `ulist` verwijdert het eerste element uit de undo-buffer met behulp van `tail` en geeft het resultaat door aan de listener `(input undo)`, en bepaalt zo de nieuwe toestand van de undo-buffer. De laatste listener uit `ulist` evalueert de toestand van het document en van de redo-buffer. De huidige waarde van het document wordt vooraan toegevoegd aan de redo-buffer. De resulterende buffer wordt doorgegeven aan de listener `(input redo)` en vormt zo de nieuwe redo-buffer.


```

        mseparator,
        mbutton [text "Quit"] quitL]),
  mcascade [text "Edit"]
    (mkMenu [] [mbutton [text "Undo"] undoL,
               mbutton [text "Redo"] redoL]),
  mcascade [text "Edit XML"]
    (mkMenuL [] (collection itemList))]

```

De mogelijke wijzigingen op een gegeven locatie in het document worden weergegeven in het `Edit XML`-menu. De inhoud van dit menu is dynamisch, want de alternatieven zijn afhankelijk van de locatie. Daarom wordt gebruik gemaakt van de functie `mkMenuL`, met `itemList` als dynamische lijst menu-items. Merk op hoe in de eerste regel `itemList` wordt geïnitieerd met één enkel menu-item. Dit item krijgt het label `<no alternatives>` en wordt in het menu getoond als er geen mogelijke wijzigingen zijn of als de gebruiker geen locatie geselecteerd heeft. Het selecteren van deze optie heeft uiteraard geen effect, de bijhorende knop wordt daarom gekoppeld aan de listener `neverL`, die elke geconsumeerde waarde negeert.

Uiteindelijk definiëren we het venster waarin de editor zal uitgevoerd worden. Hiervoor gebruiken we de functie `withRootWindow`:

```
withRootWindow :: [Conf Window] -> Component -> WComponent
```

Als configuratie-opties kunnen we onder meer met de functie `titleB` een signaal van het type `Behavior String` opgeven dat de titel van het venster zal bepalen. Hiervoor kunnen we uiteraard het corresponderende signaal van de `BVar wTitle` gebruiken. We kunnen ook het te gebruiken menu opgeven met de functie `useMenu`. Het tweede argument van `withRootWindow` is de `Component` die de eigenlijke inhoud van het venster voorstelt. Verschillende componenten kunnen samengevoegd worden door infix-operatoren zoals `above` en `beside`. In ons geval bestaat de inhoud van het venster uit twee tekstcomponenten (“edit areas”), elk voorzien van een schuifbalk (“scrollbar”):

```

v1 <- mkVScroll
v2 <- mkVScroll
str <- mkBVar ""
evb <- mkEditValB emptyEditState

```


De interpretatie van de functies `font`, `useScroll`, `width` en `height` is voor de hand liggend. De waarde `evb` heeft het type `EditValB` en stelt de toestand van de tekstcomponent voor. Een dergelijke toestand kan incrementeel gewijzigd worden door waarden van het type `EditVal` en elke toestand wordt dus bepaald door een opeenvolging van dergelijke waarden. Het abstracte type `EditValB` kunnen we dan ook beschouwen als een dynamische collectie `EditVal`-waarden. De functie `editValL` wordt gebruikt om de listener op te geven die de input-acties in de tekstcomponent, zoals het typen van een karakter, zal verwerken:

```
editValInput :: EditValB -> Listener EditVal
editValL     :: Listener EditVal -> Conf Edit
```

Hier wordt de input van de tekstcomponent dus gekoppeld aan het `evb`-object.

Met de functie `editValE` kunnen we de opeenvolging van de wijzigingen (type `Event EditVal`) opgeven die de output van de tekstcomponent zullen bepalen:

```
editValEvent :: EditValB -> Event EditVal
editValE     :: Event EditVal -> Conf Edit
```

Omdat we hier opnieuw het `evb`-object opgeven, zal alles wat de gebruiker intikt, ook onmiddellijk te zien zijn in de tekstcomponent. De functie `snapEdit` wordt gebruikt om bij elke wijziging de tekst ook te bewaren in de toestand van `str`, een `BVar` van het type `BVar String`:

```
snapEdit :: (a -> String -> b) -> Event a -> Listener b -> Conf Edit
```

Op die manier kan de gewijzigde tekst op elk moment opgevraagd worden via `str`.

De component `upperEdit` definiëren we als volgt:

```
upperEdit = mkEdit [editValE (fromListE (mapE
    (\d -> editValXml itemList evb doc str undo redo d)
    (event doc))),
    readOnly True,
    font (namedFont "Courier" 10 []),
    useScroll v1,
    width screenWidth]
```

Door de functie `readOnly` wordt vastgelegd dat de tekst in de component niet rechtstreeks door de gebruiker kan gewijzigd worden. Het heeft dan ook geen zin om een listener op te geven die de input consumeert. We gebruiken wel opnieuw `editValE` om de bron van de output voor deze tekstcomponent te bepalen. Deze output is gekoppeld aan de `BVar doc`, waarin de toestand van het document opgeslagen zit. We herinneren eraan dat het event (`event doc`) een waarde van het type `Document` genereert telkens wanneer de toestand van de onderliggende `BVar` verandert. Deze nieuwe waarde wordt dan door de functie `editValXml` omgezet naar een lijst wijzigingen van het type `[EditVal]`, met als gevolg dat de functie `mapE` het event van het type `Event Document` omzet naar een event van het type `Event [EditVal]`. De functie `fromListE` zet dit event dan om naar een event van het type `Event EditVal`, dat uiteindelijk de output van de tekstcomponent zal bepalen.

De werking van de functie `editValXml` verloopt als volgt: de functie `showXml` wordt gebruikt om de inhoud van de nieuwe documentwaarde `d` op te vragen onder de vorm van een lijst van het type `[ContentElement]`. Elke waarde van het type `ContentType` bevat alle nodige informatie om een elementair deel van het document op het scherm te tonen en er de nodige acties mee te verbinden. Zo krijgen we voor het filmdocument als eerste element uit de lijst de volgende waarde:

```
ContentElement {cpos = [], cind = 0, ctext = "film", ctype = ElementType}
```

Aan elke dergelijke waarde zal een `EditVal` gekoppeld worden. Voor het bovenstaande voorbeeld zal deze `EditVal` inhouden dat de tekst `<film>` toegevoegd wordt aan de tekstcomponent, dat deze tekst de kleur krijgt van een element (bepaald door `ctype`) en dat de tekst 0 posities inspringt (bepaald door `cind`). Als actie verbonden met het aanklikken van deze tekst, worden de alternatieven opgevraagd met de functie `alternatives` en getoond in het `Edit XML`-menu. Hiervoor kunnen we uiteraard de waarde `cpos` gebruiken, die de locatie van de tekststring in het document aangeeft. Bij het toevoegen van de menu-items zal aan elk alternatief een listener gekoppeld worden die de bijhorende wijziging doorvoert door toepassing van de functie `update`.

Hoofdstuk 4

Gerelateerd onderzoek

In dit hoofdstuk geven we een beknopt overzicht van wetenschappelijk onderzoek met betrekking tot de combinatie van XML en functionele talen. We suggereren ook hoe bepaalde toepassingen eventueel aangewend zouden kunnen worden om onze implementatie van een XML-editor uit te breiden of aan te passen.

4.1 XML en Haskell

4.1.1 HaXml

Bij de implementatie van de XML-editor werd gebruik gemaakt van HaXml. Deze toolkit vormt een implementatie van twee complementaire methodes voor het verwerken van XML in Haskell, zoals beschreven door Malcolm Wallace en Colin Runciman [38].

Een eerste aanpak is gebaseerd op een algemene boomvoorstelling van een XML-document, waarbij geen rekening gehouden wordt met de grammaticale structuur, zoals bepaald door een DTD of een schema. Er wordt een verzameling elementaire filterfuncties gedefinieerd om dergelijke bomen te verwerken. Deze eenvoudige filters kunnen ook samengesteld worden tot krachtigere operatoren met behulp van generische combinatoren. Deze combinatoren vertonen gelijkenissen met bepaalde functionalen in Funmath zoals compositie en restrictie.

Bij de tweede methode wordt wel rekening gehouden met de grammaticale structuur. Op basis van een gegeven DTD worden Haskell datatypes afgeleid voor een bepaalde klasse XML-documenten, met als gevolg dat het type-mechanisme van Haskell kan gebruikt worden om de

grammaticale correctheid van de verwerkte documenten te controleren. Zoals vermeld in hoofdstuk 1 zijn er opvallende gelijkenissen tussen de beperkte typeringstaal van DTD's en het rijkere type-systeem van Haskell. Toch treden er ook bepaalde problemen op bij het voorstellen van een DTD in Haskell, voornamelijk met betrekking tot naamgeving en hoofdlettergebruik [38].

Omdat de implementatie van de ontwikkelde editor afhankelijk is van een gegeven DTD, hebben wij uitsluitend gebruik gemaakt van de getypeerde verwerkingsmethode. Men zou in een volgend stadium echter kunnen nagaan in hoeverre de beide methodes kunnen gecombineerd worden voor bepaalde gedeeltes van de implementatie. Zo zou een combinatie van filterfuncties gebruikt kunnen worden om een bepaalde wijziging door te voeren in een document, indien men er zeker van is dat de wijziging de grammaticale correctheid niet zal verstoren. Op die manier zou men wellicht de DTD-afhankelijke code in zekere mate kunnen beperken.

Naar analogie met de specificaties in Funmath, wordt in onze implementatie de DTD rechtstreeks gebruikt — weliswaar onder de vorm van een rij typedefinities — om zowel de datatypes als de functies die inwerken op deze datatypes te genereren. Een alternatieve aanpak bestaat eruit om in eerste instantie enkel de datatypes op te stellen, en pas daarna de nodige functies te genereren, louter op basis van deze datatypes. Nader onderzoek is nodig om na te gaan of deze laatste aanpak tot elegantere code leidt.

4.1.2 Haskell XML Toolbox

De Haskell XML Toolbox [35] is een recent ontwikkelde verzameling hulpmiddelen voor het verwerken van XML met Haskell. Deze toolkit is vergelijkbaar met HaXml maar bevat ook een component voor het evalueren van XPath-uitdrukkingen. Momenteel wordt ook gewerkt aan een XSLT-implementatie. Het ontwerp van de toolkit steunt enerzijds op de filterfuncties en combinatoren uit HaXml en anderzijds op ideeën uit HXML. HXML is een in Haskell geschreven XML-parser die de geheugenvereisten minimaliseert door het veelvuldig toepassen van **lazy evaluation** en die gebruikt kan worden als vervangende component voor de parser uit HaXml.

4.2 XSLT en Haskell

Danny van Velzen beschrijft in [37] een implementatie in Haskell van een XML-bibliotheek, een XPath interpreter en een XSLT processor. XSLT is een populaire taal voor regelgebaseerde

transformaties van XML-documenten. Er wordt hierbij gebruik gemaakt van XPath om bepaalde gedeeltes in een XML-document te selecteren. Een DTD-afhankelijke XML-editor kan moeilijk rechtstreeks XSLT-transformaties toepassen, omdat deze transformaties geen rekening houden met de grammaticale structuur van XML-documenten. Men kan dus niet garanderen dat een XML-document na transformatie grammaticaal correct zal zijn met betrekking tot een gegeven DTD zonder een expliciete controle uit te voeren. In [37] wordt het ontwerp van een functie `transform :: DTD1 -> DTD2` gesuggereerd, waarbij wel rekening wordt gehouden met de grammaticale structuur, maar een implementatie van deze functie is nog niet beschikbaar. Eventueel zou de huidige XSLT-implementatie kunnen toegevoegd worden aan de editor en zou na elke transformatie het resulterende document kunnen worden opgeslagen als zuivere tekst, waarna het opnieuw geopend wordt in de editor. Bij het openen van een document wordt namelijk gecontroleerd of het document voldoet aan een gegeven DTD. Momenteel ondersteunt onze editor echter niet het simultaan gebruik van meerdere DTD's. Dit kan wellicht zonder al te veel moeilijkheden geïmplementeerd worden, maar de hinderpaal blijft dat voor elke nieuwe DTD of combinatie van DTD's de editor opnieuw moet gelinkt worden.

4.3 Web scripting in Haskell

In [29] wordt beschreven hoe men **server pages** kan construeren door Haskell te gebruiken als taal voor de ingebedde scripts. Vergelijkbare alternatieven met betrekking tot web scripting aan de server-zijde worden besproken in [30, 32, 36].

4.4 Generic Haskell

Generic Haskell [18, 19, 10, 9] is een uitbreiding van Haskell die de definitie van generische functies toelaat. De semantiek van dergelijke functies is zinvol gedefinieerd voor vrijwel alle datatypes en is enkel afhankelijk van de algemene structuur van het datatype waar de functie op inwerkt. De Generic Haskell compiler wordt gebruikt om generische functies te instantiëren voor de gewenste datatypes. De resulterende broncode kan dan gecompileerd worden met een gewone Haskell compiler. Door het gebruik van generische functies kan de hoeveelheid broncode vaak sterk beperkt worden. Omdat met elke DTD een datatype kan geassocieerd worden, kan het concept van generische functies toegepast worden voor de implementatie van DTD-afhankelijke

functionaliteit. In [23] bespreken Johan Jeuring en Paul Hagg het gebruik van Generic Haskell voor het ontwikkelen van XML-hulpmiddelen. Recent werd ook de basisfunctionaliteit van een XML-editor ontwikkeld in Generic Haskell [11, 16].

4.5 XML en Funmath

Boris Rogge introduceert in [33] het concept van functionele metadata, dat gebruikt kan worden om de functionaliteit geassocieerd met multimediapresentaties te beschrijven. Dit concept wordt gemodelleerd aan de hand van een XML-schema, waarvoor een formele beschrijving wordt opgesteld in Funmath.

Hoofdstuk 5

Conclusies

In de voorgaande hoofdstukken werd geïllustreerd hoe DTD-afhankelijke hulpmiddelen voor de verwerking van XML geïmplementeerd kunnen worden in Haskell. Dit kan vrij elegant gebeuren door de typingstaal van DTD's in te bedden in het typemechanisme van Haskell. Specifiek hebben we een structurele, grafische editor ontwikkeld voor XML-documenten. Bij het ontwerp van deze editor werd gebruik gemaakt van Funmath als brug tussen de informele beschrijving van de functionaliteit en de concrete implementatie. De formele definities vormen een implementatie-onafhankelijke karakterisatie van een structurele XML-editor. De interactie met de gebruiker werd beschreven aan de hand van de declaratieve implementatie van een GUI. De gevolgde methodiek is algemeen toepasbaar en kan als volgt samengevat worden:

- Geef een formele beschrijving van de reeds ontwikkelde programmatuur die gebruikt zal worden in de uiteindelijke implementatie. Het opstellen van deze beschrijving is meestal eenvoudig omdat de uitdrukingskracht van een formalisme als Funmath groter is dan die van bestaande programmeertalen.
- Geef een informele maar gedetailleerde beschrijving van de gewenste nieuwe functionaliteit.
- Ontwerp een formele specificatie van de gewenste functionaliteit op basis van de informele beschrijving van de nieuwe functionaliteit en de formele beschrijving van de reeds ontwikkelde programmatuur. Door de uitdrukingskracht en de universele toepasbaarheid van Funmath kan een semantische kloof tussen de informele en de formele karakterisatie grotendeels vermeden worden.

- Verifieer indien nodig bepaalde eigenschappen van de functionaliteit aan de hand van de formele definities. Funmath voorziet hiervoor een ruim gamma aan formele rekenregels.
- Implementeer een prototype op basis van de formele definities. Door de combinatie van Funmath en Haskell kan de ontwikkeling van een dergelijk prototype vrij snel en eenvoudig gebeuren. In beide talen staat het functiebegrip centraal en voor heel wat executeerbare Funmath-constructies bestaat een equivalente representatie in Haskell.
- Het ontwikkelde prototype kan indien nodig verder geoptimaliseerd worden.

We zijn van mening dat het gebruik van bovenstaande systematische ontwerpmethodede het effect van de in hoofdstuk 1 vermelde voordelen van Haskell versterkt. Zo verkleint de semantische barrière tussen de programmeur en de programmeertaal door de tussenstap van de formele karakterisatie. Als gevolg hiervan is de broncode ook duidelijker en gemakkelijker te onderhouden. Bij problemen kan de programmeur steeds teruggrijpen naar de formele definities. De productiviteit verhoogt doordat eventuele ontwerpfouten in een vroeg stadium — nog vóór de eigenlijke implementatie — kunnen ontdekt worden. De betrouwbaarheid vergroot omdat eigenschappen eenvoudiger kunnen geverifieerd worden met behulp van de formele rekenregels uit Funmath.

De ontwikkelde editor is slechts een prototype en vooral op het vlak van efficiëntie en gebruiksvriendelijkheid zijn nog verbeteringen mogelijk. Zo wordt gebruik gemaakt van een vrij eenvoudige undo-functie, die bij elke wijziging de vorige waarde van het document in een buffer opslaat. De grootte van deze buffer is uiteraard eindig, maar voor grote documenten is toch een aanzienlijke hoeveelheid geheugen nodig. Een alternatieve aanpak bestaat erin de gemaakte wijzigingen zelf bij te houden in een buffer. Op die manier kan de vorige waarde van het document geherconstrueerd worden op basis van de laatst opgeslagen waarde en de daarna uitgevoerde wijzigingen. Voor grote documenten geniet deze laatste aanpak wellicht de voorkeur, omdat de extra uitvoeringstijd voor het herconstrueren van documentwaarden minder doorweegt dan de uitgespaarde geheugenruimte.

Bij de ontwikkeling van de XML-editor hebben we gebruik gemaakt van de HaXml-toolkit. Gezien de beschikbare tijd voor de realisatie van dit afstudeerwerk, was dit een voor de hand liggende keuze. Door het hergebruik van deze programmatuur, kon het vereiste implementatiewerk in grote mate beperkt worden. Een nadeel van deze aanpak is dat bepaalde tekortkomingen van HaXml moeilijk konden weggewerkt worden in de implementatie van de editor. De ontwikkeling van een vergelijkbare editor zonder het gebruik van HaXml of een gelijkaardige toolkit

vergt naar onze mening echter een aanzienlijk groter project.

Als eerste beperking van HaXml vermelden we het ontbreken van ondersteuning voor XML Namespaces. HaXml voldoet ook niet volledig aan de XML 1.0 standaard [37].

Belangrijker is dat HaXml enkel gebruik maakt van DTD's en dus niet compatibel is met XML Schema. Naar we weten is er geen vergelijkbare toolkit voor Haskell beschikbaar die wel werkt met XML Schema. De ontwikkeling van een dergelijke toolkit is niet voor de hand liggend, omdat de typeringstaal van schema's veel rijker is dan die van DTD's en bijgevolg niet op dezelfde eenvoudige manier kan ingebed worden in Haskell. Een implementatie van XML Schema in Haskell vereist het realiseren van een diepere inbedding (deep embedding), waarbij Haskell gebruikt wordt om de datatypes uit de typeringstaal van XML Schema te definiëren. Bij de verwerking van DTD's is deze tussenstap niet nodig, omdat de datatypes uit de typeringstaal van DTD's een deelverzameling vormen van de reeds in Haskell ingebouwde datatypes. Dit verklaart wellicht waarom het gebruik van DTD's in de Haskell-wereld nog steeds bijzonder populair is [38, 35, 19, 23].

We vermelden ook het probleem van elegante foutafhandeling. Als een bepaalde component van HaXml, zoals de XML-parser, een ongeldig document te verwerken krijgt, dan stopt het programma met het uitprinten van een foutmelding. In een editor is het echter niet wenselijk dat het programma beëindigd wordt bij het openen van een dergelijk document. Om dit probleem op te vangen moeten eerst bepaalde componenten van HaXml moeten herschreven worden. Zo kan de functie `readXml`, met als resultaattype `IO a` vervangen worden door een functie met als resultaattype `IO (Maybe a)`. Deze nieuwe functie genereert dan geen foutmelding bij het inlezen van een ongeldig XML-document, maar geeft in plaats daarvan de waarde `Nothing` als resultaat terug. In de editor kan dan bij het inlezen van de waarde `Nothing` aan de gebruiker gemeld worden dat het ingelezen document geen geldig XML-document is. Een meer geavanceerde methode van foutafhandeling wordt besproken in [26].

We merken op dat de toolkit `FranTk` — gebruikt voor de implementatie van de grafische gebruikersinterface — in een vrij vroege ontwikkelingsfase zit en bovendien schaars gedocumenteerd is. Bepaalde aspecten van de functionaliteit zijn dan ook in dit stadium moeilijk of niet te implementeren. Zo zal de tekstcomponent uit de editor na elke wijziging het gewijzigde document weergeven vanaf de eerste regel. Een typische gebruiker verwacht wellicht dat het gewijzigde document in een dergelijk geval weergegeven wordt vanaf de locatie waar de wijziging plaats vond, maar dit lijkt ons met de beschikbare functionaliteit voor tekstcomponenten momenteel

niet te implementeren.

Ondanks de genoemde beperkingen kunnen we toch stellen dat het ontwikkelde prototype de gewenste functionaliteit met bescheiden middelen realiseert. De implementatie van de kernfunctionaliteit van de editor bestaat uit niet meer dan 350 regels broncode. Voor de implementatie van de GUI zijn minder dan 200 regels broncode nodig. De implementatie van de gebruikte toolkits — HaXml en FranTk — wordt hier niet meegerekend, maar dat is uiteraard een van de voordelen van hergebruik van programmatuur.

Referenties

- [1] S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman en S. Zilles. “Extensible Stylesheet Language (XSL) Version 1.0”. W3C Recommendation, 2001
(<http://www.w3.org/TR/xsl/>)
- [2] R. Bird. *Introduction to Functional Programming using Haskell: second edition*. Prentice Hall, 1998
- [3] R. T. Boute. “Concrete Generic Functionals: Principles, Design and Applications”. *Proceedings IFIP Conference on Generic Programming*, 2003, 89-119
- [4] R. T. Boute. “Formal logic for practical use: a calculational approach”. cursusnota’s Formele Semantiek en Formele Systeemmodellen, Universiteit Gent, 2002
- [5] R. T. Boute. “Functional Mathematics: a Unifying Declarative and Calculational Approach to Systems, Circuits and Programs — Part I”. cursusnota’s Basiswiskunde voor Computerwetenschappen, Universiteit Gent, 2002
- [6] T. Bray, D. Hollander en A. Layman. “Namespaces in XML”. W3C Recommendation, 1999
(<http://www.w3.org/TR/REC-xml-names/>)
- [7] T. Bray, J. Paoli, C. M. Sperberg-McQueen en Eve Maler. “Extensible Markup Language (XML) 1.0 (Second Edition)”. W3C Recommendation, 2000
(<http://www.w3.org/TR/REC-xml/>)
- [8] J. Clark en S. DeRose. “XML Path Language (XPath) Version 1.0”. W3C Recommendation, 1999
(<http://www.w3.org/TR/xpath/>)

-
- [9] D. Clarke, J. Jeuring en A. Löh. “The Generic Haskell User’s Guide”. Technical Report UU-CS-2002-047, Universiteit Utrecht, 2002
- [10] D. Clarke en A. Löh. “Generic Haskell, Specifically”. *Proceedings IFIP Conference on Generic Programming*, 2003, 21-47
- [11] J. de Wit. “A technical overview of Generic Haskell”. afstudeerwerk, Universiteit Utrecht, 2002
- [12] S. DeRose, R. Daniel Jr., P. Grosso, E. Maler J. Marsh en N. Walsh. “XML Pointer Language (XPointer)”. W3C Working Draft, 2002
(<http://www.w3.org/TR/xptr/>)
- [13] S. DeRose, E. Maler en D. Orchard. “XML Linking Language (XLink) Version 1.0”. W3C Recommendation, 2001
(<http://www.w3.org/TR/xlink/>)
- [14] C. Elliott en P. Hudak. “Functional Reactive Animation”. *Proceedings ACM SIGPLAN International Conference on Functional Programming*, 1997, 263-273
- [15] D. C. Fallside. “XML Schema Part 0: Primer”. W3C Recommendation, 2001
(<http://www.w3.org/TR/xmlschema-0/>)
- [16] P. Hagg. “A framework for developing generic XML tools”. afstudeerwerk, Universiteit Utrecht, 2002
- [17] E. R. Harold. *XML Bible*. Hungry Minds, 1999
- [18] R. Hinze en J. Jeuring. “Generic Haskell: Practice and Theory”. *Lecture Notes of the Summer School on Generic Programming*, Springer-Verlag, 2003
- [19] R. Hinze en J. Jeuring. “Generic Haskell: Applications”. *Lecture Notes of the Summer School on Generic Programming*, Springer-Verlag, 2003
- [20] P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000
- [21] D. Hunter, J. Rafter, J. Pinnock, C. Dix, K. Cagle en R. Kovack. *Beginning XML*. Wrox Press, 2001

- [22] P. Hudak, J. Peterson en J. Fasel. “A Gentle Introduction to Haskell”. 2000
(<http://www.haskell.org/tutorial/>)
- [23] J. Jeuring en P. Hagg. “Generic Programming for XML Tools”. Technical Report UU-CS-2002-023, Universiteit Utrecht, 2002
- [24] S. P. Jones. “A Short Introduction to Haskell”. 2001
(<http://www.haskell.org/aboutHaskell.html>)
- [25] S. P. Jones, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, J. Hughes, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman en P. Wadler. “Haskell 98 Language and Libraries: The Revised Report”. 2002
(<http://www.haskell.org/onlinereport/>)
- [26] S. P. Jones. “Tackling the Awkward Squad: monadic input/output, concurrency, exceptions and foreign-language calls in Haskell”. *Engineering theories of software construction*, IOS Press, 2001, 47-96
- [27] S. S. Laurent. *XML: A Primer*. Hungry Minds, 1999
- [28] K. Luyten. “Inleiding tot XML”. cursusnota’s Technologie van Multimediasystemen en Software, Limburgs Universitair Centrum, 2002
- [29] E. Meijer en D. van Velzen. “Haskell Server Pages: Functional Programming and the Battle for the Middle Tier”. *Proceedings Haskell Workshop*, 2000
- [30] E. Meijer. “Server side web scripting in Haskell”. *Journal of Functional Programming*, 10(1), 2000, 1-18
- [31] E. Meijer, D. Leijen en J. Hook. “Client-side Web Scripting with HaskellScript”. *Proceedings PADL*, 1999
- [32] E. Meijer en J. van Dijk. “Perl for swine: CGI programming in Haskell”. *Proceedings First Workshop on Functional Programming*, 1996
- [33] B. Rogge. “Functionele metadata: een softwareraamwerk voor het opzetten van multimedia-toepassingen”. doctoraatswerk, Universiteit Gent, 2002

-
- [34] M. Sage. “FranTk — A Declarative GUI System for Haskell”. *Proceedings ACM SIGPLAN International Conference on Functional Programming*, 2000
- [35] M. Schmidt. “Design and Implementation of a validating XML parser in Haskell”. afstudeerwerk, Universiteit Wedel, 2002
- [36] P. Thiemann. “Wash/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms”. *Practical Aspects of Declarative Languages*, 2002
- [37] D. van Velzen. “An XSLT implementation in Haskell”. afstudeerwerk, Universiteit Utrecht, 2001
- [38] M. Wallace en C. Runciman. “Haskell and XML: Generic Combinators or Type-Based Translation?”. *Proceedings International Conference on Functional Programming*, 1999
- [39] H. Williamson. *XML: The Complete Reference*. McGraw-Hill Osborne Media, 2001

