4.8.2 Anwendung: Analyse von Zugangskontrollen

Um die Sicherheit von IT-Systemen zu gewährleisten, werden oft Richtlinien zur Zugangskontrolle ("access control policies") definiert. Darin ist z.B. festgelegt, welche Gruppen von Personen zu welchen Komponenten eines Systems Zugang haben und was sie dort machen dürfen. Diese Richtlinien kann man durch formale Regeln definieren, die dann in einem System geprüft werden. Wenn diese Regeln verändert werden, stellt sich die Frage nach den Konsequenzen dieser Änderungen, z.B. wer durch bestimmte Regeländerungen Zugang erhält. In [Bertolissi/Talbot/Villevalois 16] haben die Autoren Zugangsrichtlinien durch Termersetzungssysteme modelliert und dann Needed Narrowing verwendet, um solche Fragestellungen zu beantworten. Diesen Ansatz kann man sehr einfach in Curry umsetzen, wie im folgenden gezeigt wird.
Die Definition von Zugangskontrollregeln verlangt einige Basisbegriffe, die für ein bestimmtes System weiter konkretisiert werden. In diesem Fall sind dies:

`Principal`: Eine Objekt (Person oder auch Programm), das etwas mit dem IT-System machen will.

`Resource`: Ein Element des IT-Systems, mit dem ein `Principal` etwas tun will.

`Action`: Eine Methode zum Zugriff auf eine `Resource`.

`Authorization`: Antworten auf den Wunsch, auf eine `Resource` zuzugreifen, d.h. `Grant` oder `Deny`.

`Category`: Eine Klassifizierung von Personen, Rollen u.ä., die auch hierarchisch erfolgen kann.

In unserem IT-System nehmen wir an, dass diese Begriffe die folgenden konkreten Werte haben können:

```
data Category = Administrative | Accounting | Sales
 deriving (Eq,Show)

data Principal = Alice | Bob | Carol
 deriving (Eq,Show)

data Action = Edit | View
 deriving (Eq,Show)

data Resource = PasswdFile | SalesDB | AccountingDB
 deriving (Eq,Show)

data Authorization = Grant | Deny
 deriving (Eq,Show)
```

Um diese Objekte in Beziehung zu setzen, werden einige Funktionen definiert.

Die Hierarchie von Kategorien wird durch eine Operation `subCats` definiert, die zu einer Kategorie die darin enthaltenen Kategorien angibt:

```
subCats :: Category → [Category]
subCats Administrative = [Accounting, Sales]
subCats Accounting     = []
subCats Sales          = []
```

Für jeden `Principal` wird mittels der Operation `roles` definiert, zu welchen Kategorien dieser gehört:

```
roles :: Principal → [Category]
roles Alice = [Administrative]
roles Bob   = [Sales]
roles Carol = [Accounting]
```

Zur Definition, wer auf was zugreifen darf ("permissions"), wird jeder Kategorie eine Liste von Paaren `Action`/`Resource` zugeordnert:

```
permits :: Category → [(Action,Resource)]
permits Administrative = [(Edit, PasswdFile),   (View, PasswdFile)]
permits Accounting     = [(Edit, AccountingDB), (View, SalesDB)]
permits Sales          = [(Edit, SalesDB),      (View, AccountingDB)]
```

Nun können wir die Autorisierung, d.h. die Frage, ob ein `Principal` eine `Action` auf einer `Resource` ausführen kann, durch eine weitere Operation `auth` definieren. Axiomatisch ist dies definiert als: falls ein `Principal` direkt oder indirekt zu einer Kategorie gehört, die die Erlaubnis hat, die `Action` auf einer `Resource` auszuführen, dann ist die Autorisierung gegeben. Wir können dies wie folgt in Curry definieren:

```
auth :: Principal → Action → Resource → Authorization
auth p a r | (a, r) `elem` concatMap permits (allCats (roles p)) = Grant
           | otherwise                                           = Deny
 where
  allCats (c : cs) = c : allCats (subCats c ++ cs)
  allCats []       = []
```

Mittels Narrowing können wir nun z.B. abfragen, welche Dinge eine bestimmte Person verändern darf:

```
> solve $ auth Alice Edit x == Grant where x free
{x=PasswdFile} True
{x=SalesDB} True
{x=AccountingDB} True
```

Wir können auch abfragen, welche Personen eine bestimmte `Resource` verändern dürfen:

```
> solve $ auth x Edit SalesDB == Grant where x free
{x=Alice} True
{x=Bob} True
```

Manchmal möchte der Systemadministrator die Zugangsregeln verändern. Z.B. könnte er festlegen, dass von nun an Benutzer mit Administratorrechten die Rolle `Accounting` nicht mehr haben, in dem die erste Gleichung für `subCats` wie folgt verändert wird:

```
subCats Administrative = [Sales]
...
```

Hierdurch verliert z.B. `Alice` die Möglichkeit, `AccountingDB` zu verändern:

```
> solve $ auth Alice Edit x == Grant where x free
{x=PasswdFile} True
{x=SalesDB} True
```

Interessanter ist aber die Frage, welche Konsequenzen die Regeländerungen insgesamt haben, d.h. welche Autorisierungen verändert werden. Auch dies können wir mittels Narrowing berechnen. Hierzu ist es notwendig, verschiedene Zugriffskontrollregeln miteinander zu vergleichen, d.h. wir müssen mehrere Zugriffskontrollregeln gleichzeitig betrachten. Zu diesem Zweck parametrisieren wir die Autorisierungsoperation `auth` über die Operationen, die die Zugriffskontrollregeln spezifizieren:

```
authP :: (Category  → [(Action,Resource)])
       → (Principal  → [Category])
       → (Category  → [Category])
       → Principal  → Action  → Resource  → Authorization
authP car pc cc p a r
  | (a, r) `elem` concatMap car (allCats (pc p)) = Grant
  | otherwise                                    = Deny
 where
  allCats :: [Category]  → [Category]
  allCats (c : cs) = c : allCats (cc c ++ cs)
  allCats []       = []
```

Weiterhin definieren wir eine separate Operation für die Veränderungen an der ursprünglichen Operation `subCats`:[14]

```
subCats' :: Category  → [Category]
subCats' Administrative = [Sales]
subCats' Accounting     = []
subCats' Sales          = []
```

[14]Falls wir an den anderen Operationen auch etwas ändern, müssten wir hierfür auch neue Varianten definieren.

Nun können wir einfach eine Operation definieren, die für ein Objekt, eine Aktion und eine Ressource die Unterschiede in den Autorisierungen bezüglich der alten und neuen Regeln berechnet:

```
diffAuths :: Principal  → Action  → Resource
           → (Authorization, Authorization)
diffAuths p a r
  | authP permits roles subCats  p a r == auth1 &&
    authP permits roles subCats' p a r == auth2 &&
    auth1 /= auth2
  = (auth1, auth2)
 where
  auth1,auth2 free
```

Damit können wir alle Unterschiede zwischen diesen Zugriffsregeln wie folgt berechnen:

```
> diffAuths p a r   where p,a,r free
{p=Alice, a=Edit, r=AccountingDB} (Grant,Deny)
{p=Alice, a=View, r=SalesDB} (Grant,Deny)
```

Als Ergebnis sehen wir, dass `Alice` z.B. den Zugriff zum Verändern der `AccountingDB` verliert.

Auf Grund der Korrektheit und Vollständigkeit von Narrowing werden durch dieses Programm alle Konsequenzen der Zugangsregeländerungen berechnet [Bertolissi/Talbot/Villevalois 16].

4.8.3 XML-Programmierung mit funktionalen Mustern

XML ist heutzutage ein wichtiges Format zum Austausch von Dokumenten. XML ist eine sogenannte **Auszeichnungssprache** (**markup language**), mit der man in Dokumenten bestimmte Strukturen deutlich machen kann. Die Auszeichnungselemente haben einen Namen (tag), der in spitzen Klammern eingefasst wird. Beispielsweise könnte das folgende XML-Dokument ein Auszug aus einer Kontaktliste sein:

```
<contacts>
  <entry>
    <name>Hanus</name>
    <first>Michael</first>
    <phone>+49-431-8807271</phone>
    <email>mh@informatik.uni-kiel.de</email>
    <email>hanus@email.uni-kiel.de</email>
  </entry>
  <entry>
    <name>Smith</name>
    <first>William</first>
    <nickname>Bill</nickname>
```

```
    <phone>+1-987-742-9388</phone>
  </entry>
</contacts>
```

Der Vorteil dieses Formats liegt in dem standardisierten Austausch und der einheitlichen Verarbeitung der Dokumente. Zum Beispiel kann ein Datenbanksystem die Ergebnisse einer Anfrage im XML-Format exportieren, sodass andere Programme diese einfach einlesen und die Strukturen weiter verarbeiten können. Beispielsweise liefert das an der CAU Kiel eingesetzte UnivIS seine Daten auch im XML-Format aus.

In der Praxis können dabei allerdings Schwierigkeiten auftreten, insbesondere wenn die konkreten Datenformate eine komplizierte Struktur besitzen oder sich die Struktur über die Zeit ändert:

- Obwohl für viele Bereiche konkrete XML-Sprachen definiert sind, sind diese oft sehr komplex, sodass es aufwändig ist, diese zu verarbeiten, wenn man nur an einigen Teildaten interessiert ist.
- Manchmal gibt es auch keine präzise XML-Spezifikation oder diese verändert sich mit der Einsatzzeit der Systeme, sodass man die zugehörigen Programme zur XML-Verarbeitung anpassen muss.

Bei Vorliegen des letzten Punktes spricht man daher auch von „semi-strukturierten“ Daten. Dies ist z.B. bei dem obigen Beispieldokument der Fall: der erste Eintrag hat zwei E-Mail-Adressen und keinen Spitznamen, während der zweite Eintrag keine E-Mail-Adresse, dafür aber einen Spitznamen hat. Bei der präzisen Verarbeitung dieser Daten, z.B. mittels Pattern Matching, müsste man alle diese Fälle abdecken.

Um dem Programmierer die Arbeit zu erleichtern, gibt es verschiedene Sprachen und Werkzeuge zur Verarbeitung von XML-Dokumenten, wie z.B.

- XPath[15]: Pfadausdrücke zur Adressierung von Teildokumenten (wie navigiere ich von der Wurzel zum gewünschten Teilausdruck?)
- XQuery[16]: Selektion von XML-Dokumenten aufbauend auf XPath
- XSLT[17]: Transformation von XML-Dokumenten aufbauend auf XPath

All dies sind mächtige Werkzeuge, aber wegen der Verwendung von XPath eher imperativ (*wie* gehe ich...?) und für einfache Anwendungen manchmal zu aufwändig zu benutzen.

Als weitere Alternative wurde die Sprache Xcerpt [Bry/Schaffert 02] vorgeschlagen, die auf Ideen der Logikprogrammierung basiert und Pattern Matching auf partiellen Datentermen (d.h. Termen, deren Struktur nicht vollständig spezifiziert wird) erlaubt. Nachfolgend wollen wir zeigen, wie man eine ähnliche Idee mit geringem Aufwand mittels funktionaler Muster umsetzen kann.

[15] `http://www.w3.org/TR/xpath`
[16] `http://www.w3.org/XML/Query/`
[17] `http://www.w3.org/TR/xslt`

XML-Dokumente sind nichts anderes als hierarchische Datenstrukturen, d.h. Bäume bzw. Termstrukturen. Daher kann man diese einfach durch den folgenden Datentyp repräsentieren:

```
data XmlExp = XText String
            | XElem String [(String,String)] [XmlExp]
```

Eine XML-Dokument oder XML-Ausdruck (`XmlExp`) ist somit entweder ein Text oder eine Struktur mit einem Namen, Attributen (Liste von Stringpaaren) und einer Liste von darin enthaltenen XML-Dokumenten (andere spezielle XML-Elemente werden hier der Einfachheit halber ignoriert).

Um XML-Dokumente in einem Programm einfacher aufzuschreiben, definieren wir folgende Abstraktionen:

```
xtxt :: String → XmlExp
xtxt s = XText s

xml :: String → [XmlExp] → XmlExp
xml t xs = XElem t [] xs
```

Hiermit können wir nun die zweite `entry`-Struktur aus dem obigen Beispiel als folgenden Term definieren:

```
xml "entry" [xml "name"     [xtxt "Smith"],
             xml "first"    [xtxt "William"],
             xml "nickname" [xtxt "Bill"],
             xml "phone"    [xtxt "+1-987-742-9388"]]
```

Das Curry-Modul `XML`[18] enthält diese Definitionen zusammen mit weiteren nützlichen Operationen, z.B. zum Lesen von XML-Textdokumenten:

```
parseXmlString :: String → [XmlExp]
readXmlFile    :: String → IO XmlExp
```

[18] Die Bibliothek `XML` befindet sich im Paket `xml`, welches mit Hilfes des Curry Package Managers durch "`cypm add xml`" installiert werden kann.