Inferência de tipos em documentos XML

José João Almeida and Alberto Manuel Simões

Departamento de Informática, Universidade do Minho
{jj|ambs}@di.uminho.pt

Resumo A estrutura dos documentos XML é descrita, habitualmente, em DTDs e/ou Schemas, o que permite ao programador estudar a forma de processamento estrutural mais correcta para o tipo de documento em causa.
No entanto, outros documentos há em que o tipo de documento não está definido e que obriga a analisar o documento para inferir a estrutura em causa. Paralelamente algumas especificações baseadas em Schemas tendem a ser de tal modo grandes que se tornam impossíveis de ler.
Neste documento pretendemos apresentar a ferramenta PFS[1], capaz de inferir tipos a partir de documentos XML, de mostrar de forma compacta essa informação e pretendemos ainda mostrar como esses tipos podem ajudar no processamento desses documentos (usando XML::DT com tipos).

1 Introdução

Como o XML é uma linguagem fortemente estruturada, a forma mais elegante de processar documentos XML é baseada na estrutura do documento. A estrutura de um documento não é mais do que um mapeamento de cada elemento a um tipo ou estrutura de dados.

Considere como exemplo o elemento `ul` do `XHTML`[**?**]. A estrutura de dados a si associada é uma lista, já que os seus filhos são etiquetas de um mesmo tipo (`li`).

Da mesma forma, ao elemento `item` do `RSS/RDF`[**?**] o tipo de dados associado é uma função finita, já que todos os seus filhos têm nomes diferentes e não são mais do que campos do elemento `item`.

Estes dois exemplos fazem parte de duas especificações da W3C que, como tal, estão bem documentadas e incluem DTDs e Schemas associados. No entanto, documentos há em que não existe um DTD associado. Para estes, é importante a existência de uma ferramenta que permita inferir os tipos e estruturas associadas a cada elemento.

Sem olhar para um DTD ou Schema, o tipo de um documento XML pode ser descrito formalmente por:

$$\begin{aligned} XML &\equiv elemento \\ elemento &\equiv nome \times atributos \times filho^\star \\ atributos &\equiv nomeAtributo \mapsto valorAtributo \\ filho &\equiv texto + elemento \end{aligned}$$

[1] *Pig from sausages.*

Esta estrutura arbórea permite descrever sintacticamente a informação, mas não lhe associa semântica.

Imagine-se a representação de uma lista de elementos em XML. A representação típica é:

```
<!ELEMENT lista (item*)>
```

onde aparece uma etiqueta externa (lista) e uma para cada elemento (item). Outros exemplos incluem listas directamente como filhos de elementos (sem a etiqueta externa):

```
<!ELEMENT data (a, b, c, item*, d, e)>
```

Da mesma forma, há uma série de propriedades de estrutura que não têm fácil expressão: a simples estrutura não nos diz se esta lista poderá ter elementos repetidos (se será um conjunto).

A definição de uma função finita[2] também pode tomar várias formas. Quando o domínio é fechado e de baixa cardinalidade, o mais simples será usar um conjunto de etiquetas (sem repetições) que servem de chaves:

```
<!ELEMENT hash (a, b, c, d, e, f, g)>
```

No entanto, nada obsta a que se defina funções finitas com outras sintaxes:

1.
```
<!ELEMENT hash (item+)>
<!ELEMENT item (#PCDATA)>
<!ATTLIST item key CDATA #REQUIRED>
```

2.
```
<!ELEMENT hash (key, data)+>
```

3.
```
<!ELEMENT hash (item+)>
<!ELEMENT item EMPTY>
<!ATTLIST item key  CDATA #REQUIRED
               data CDATA #REQUIRED>
```

Com toda esta panóplia de possíveis combinações de etiquetas, torna-se possível apresentar de formas muito diversas várias estruturas de dados simples. Por outro lado, torna-se impraticável detectar (adivinhar) automaticamente todos estes tipos de dados.

No entanto há várias situações que podem ser tratadas de modo homogéneo e automático, levantando duas questões diferentes:

- a sua descrição e processamento,
- a sua detecção automática

[2] Designaremos por **Função Finita de A para B**, as correspondências unívocas de A para B. Esta designação inclui as HASH do Perl, os Arrays associativos, os Mappings, etc.

1.1 Tipos de elementos

Esta secção discute alguns tipos de dados que conseguimos encontrar em documentos XML cuja detecção automática é viável.

Cada tipo corresponde a uma combinação de etiquetas XML que pretendemos ver tratadas como um todo, e corresponde ainda a uma estratégia de processamento dos filhos (tipicamente a um agrupar numa estrutura única os resultados dos processamentos dos filhos).

- **String** (STR)
  A maior parte dos elementos mistos irão, com certeza, cair neste tipo. Por exemplo, o elemento `p` do `XHTML` é um elemento misto cujos filhos são, por exemplo, `b`, `i` e outros, que deverão ser concatenados numa string depois de processados.

- **Sequência** (SEQ)
  Alguns elementos servem de delimitadores de uma lista de filhos, todos com o mesmo nome. Este é um caso bastante típico, e ao qual associamos o tipo *sequência*.
  Por exemplo, os elementos `ul` ou `ol` são do tipo *sequência*, já que são constituídos por uma lista de items (`li`).
  Um processador típico para este tipo de elemento terá de lidar com uma lista ou sequência de elementos.

- **Mapeamento** (MAP)
  Existem elementos que são delimitadores de um conjunto de vários elementos, todos com nomes diferentes. Por exemplo, o elemento `item` do `RSS/RDF` é um destes casos:

```
<item rdf:about="http://localhost/archives/2004/11/syndication.html">
  <title>Syndication</title>
  <link>http://localhost/archives/2004/11/syndication.html</link>
  <description>
     <![CDATA[<p>I forgot to told you, my dear readers, that this
      blog is syndicated on <a href="http://planet.botfu.org">Planet
      BotFu</a>, where all the folks from <tt>#botfu</tt> blog their
      lifes.</p>]]>
  </description>
  <dc:subject>Misc</dc:subject>
  <dc:creator>null</dc:creator>
  <dc:date>2004-11-12T11:05:52+00:00</dc:date>
</item>
```

- **Mapeamento Múltiplo** (MULTIMAP)
  Há casos em que um elemento contém várias etiquetas, mas cujos nomes se vão repetindo. Este modelo cria um mapeamento do nome da etiqueta para uma lista dos conteúdos de todas as etiquetas com esse nome.
  Ou seja, para um XML como

```
<exemplo>
  <bar> textoA </bar>
  <foo> textoB </foo>
  <bar> textoC </bar>
  <bar> textoD </bar>
  <foo> textoE </foo>
</exemplo>
```

construiríamos uma árvore como a seguinte:

```
{
  bar => ["textoA", "textoC", "textoD"],
  foo => ["textoB", "textoE"],
}
```

– **Mapeamento Múltiplo Selectivo** (MMAPON)
Este modelo de dados é o meio termo entre o mapeamento simples e o mapeamento múltiplo, permitindo ao utilizador especificar quais os elementos que vão ter repetições.
Ou seja, para um XML como

```
<exemplo>
  <bar> textoA </bar>
  <foo> textoB </foo>
  <bar> textoC </bar>
  <bar> textoD </bar>
</exemplo>
```

e usando o modelo `MMAPON(bar)`, construiríamos uma árvore como a seguinte:

```
{
  bar => ["textoA", "textoC", "textoD"],
  foo => "textoB",
}
```

– **Elemento Único** (THECHILD)
Embora pouco provável, especialmente em documentos bem estruturados, é possível em alguns formatos a existência de um elemento com apenas um filho...

2 XML::DT baseado em tipos

Embora o conhecimento dos tipos associados a cada elemento possa ajudar o programador na sua tarefa, este trabalho está ainda mais simplificado ao usar uma ferramenta que consiga tirar partido da associação de tipos aos elementos XML, como é o caso do XML::DT[**?**,**?**].

Antes da apresentação de como se pode inferir os tipos de documentos XML, torna-se imperativa a apresentação de alguns exemplos de como o XML::DT lida com eles.

2.1 Exemplo introdutório

Considere-se, por exemplo, o seguinte documento XML:

```
<?xml version="1.0"?>
<institution>
  <id>U.M.</id>
  <name>University of Minho</name>
  <tels>
    <item>1111</item>
    <item>1112</item>
    <item>1113</item>
  </tels>
  <where>Portugal</where>
  <contacts>J.Joao; J.Rocha; J.Ramalho</contacts>
</institution>
```

Neste exemplo, `institution` pode ser vista como uma função finita de elemento para o seu conteúdo, já que cada etiqueta está a definir propriedades (sem haver propriedades repetidas) de um mesmo objecto (uma instituição). Por outro lado, a etiqueta `tels` agrega uma lista de telefones.

Assim, um processador Perl usando o XML::DT para processar este documento pode ser escrito como:

```
#!/usr/bin/perl -w
use XML::DT;

%handler = ( -default => sub{$c},
             -type    => { institution => 'MAP',
                           tels        => 'SEQ' },
             contacts => sub{ [ split(";",$c)] },
           );

$a = dt("ficheiro.xml", %handler);
```

Este exemplo define que, por omissão (regra `-default`), o processamento de uma etiqueta é, simplesmente, devolver o seu conteúdo (`$c` - guarda o *contents*).

Por sua vez, que os tipos (regra `-type`) das etiquetas `institution` e `tels` são, respectivamente, uma função finita e uma sequência.

A regra seguinte, para a etiqueta `contacts` divide o seu conteúdo pelo caracter ponto-e-vírgula (split) retornando a lista de contactos.

O processamento do documento apresentado com este pequeno código, gera a seguinte estrutura de dados Perl:

```
{
  tels     => [ 1111, 1112, 1113 ],
  name     => 'University of Minho',
  where    => 'Portugal',
  id       => 'U.M.',
  contacts => [ 'J.Joao', ' J.Rocha', ' J.Ramalho' ],
}
```

2.2 Exemplo RSS

Para demonstrar a versatilidade do uso de tipos para o processamento de documentos XML, consideremos a transformação de um documento RSS para HTML.

```
#!/usr/bin/perl
use CGI qw/:standard/;
use XML::DT;

%hdl=(
    -default => sub {$c},

    -type => { channel   => 'MAP',
               item      => 'MAP', },

    'channel' => sub{
      h1(a({-href => $c->{link}},
           img({-src => $c->{image}}), $c->{title}))
    },

    'item' => sub {
      h3(a({-href => $c->{link}},$c->{title})).
        blockquote($c->{description})
    },

);

print dt(shift(), %hdl);
```

A estrutura do processador é semelhante à do exemplo anterior, com a definição de que os elementos `channel` e `item` são tratados como funções finitas. O primeiro elemento, define as propriedades do canal RSS que estamos a transformar, e cada `item` é uma das notícias/entradas do RSS.

As respectivas entradas para os elementos `channel` e `item` são funções de processamento que recebem como *contents* (`$c`) uma função finita (hash) e constroem o HTML respectivo.

3 Inferência de tipos com PFS

A inferência de tipos XML::DT partindo de um DTD (ou Schema) é quase directa mas, como vimos, nem sempre temos o DTD para analisar. O nosso objectivo é inferir tipos directamente do documento XML, e criar automaticamente um conjunto de vistas que permitam ao utilizador interpretar a estrutura do documento XML.

O Trang[**?**] é uma ferramenta desenvolvida por James Clark que converte DTD em Schemas e vice-versa, bem como noutros formatos. Uma das suas potencialidades é a adivinhação de um DTD ou Schema a partir de um documento XML. No entanto, o PFS pretende ser mais do que isso, já que deverá ser capaz de obter informação mais detalhada do que a definida num DTD ou Schema.

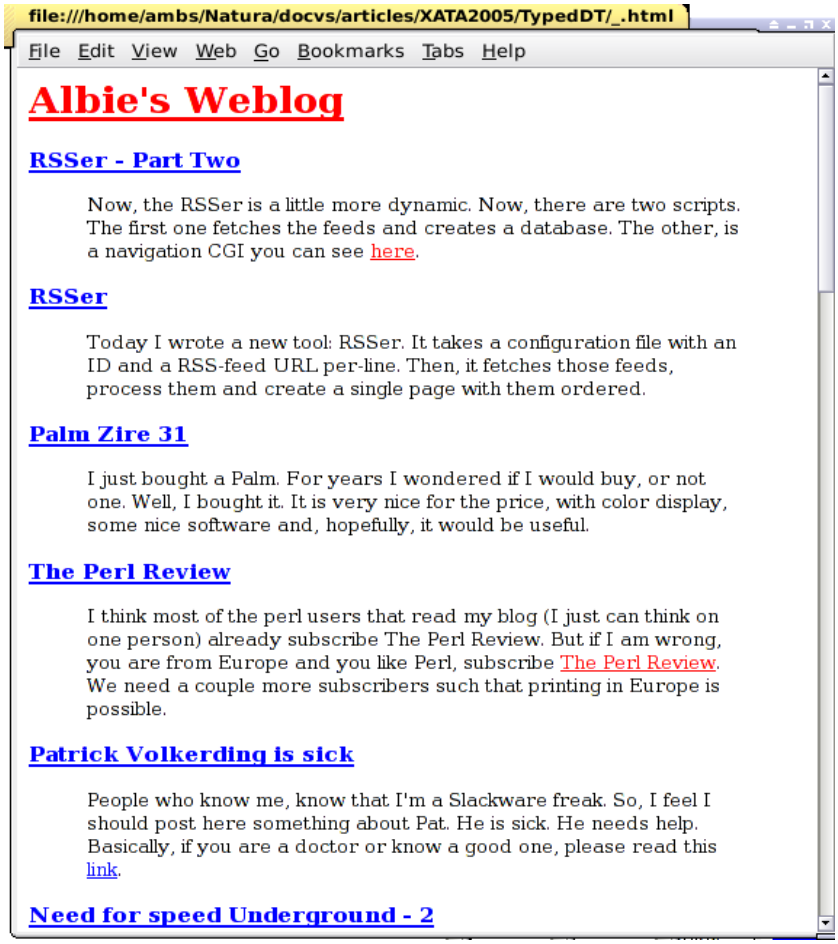

Figura 1. HTML Gerado de um documento RSS

3.1 Uso do PFS

Como primeiro exemplo, consideremos e exemplo do XML que define uma lista de números de telefone, que vimos anteriormente. O resultado do PFS sobre este documento devolve o seguinte resumo:

```
# institution ...Tue Nov 30 15:30:03 2004
institution  =>  tup(contacts, name, tels, id, where)
contacts     =>  text
name         =>  text
tels         =>  seq(item)
id           =>  text
where        =>  text
item         =>  text
```

Neste resumo da estrutura do documento são explicitados os elementos de texto (`#PCDATA`) bem como que o elemento `tels` contém uma sequência de items, e o `institution` é um tuplo de elementos (identificados pelo nome da respectiva etiqueta). Como vimos, a forma natural de representar este tuplo em Perl será o uso de uma função finita de nome de elemento para o seu conteúdo.

O resultado de analisar um documento RSS é um pouco mais complicado:

```
# rdf:RDF ...Tue Nov 30 15:46:19 2004
rdf:RDF     =>  mtup(channel, item+)
item        =>  tup(link, dc:creator, dc:subject, title, description,
                    dc:date) * rdf:about
channel     =>  tup(link, dc:creator, title, admin:generatorAgent,
                    description, dc:date, items) * rdf:about
dc:creator  =>  text
link        =>  text
dc:subject  =>  text
title       =>  text
dc:date     =>  text
description =>  text
admin:generatorAgent =>  empty * rdf:resource
items       =>  rdf:Seq
rdf:Seq     =>  seq(rdf:li)
rdf:li      =>  empty * rdf:resource
```

Neste exemplo já é possível ver alguns elementos com atributos (separados da definição do tipo de elemento por um asterisco) e alguns tipos mais complicados:

- o elemento `rdf:RDF` é um mapeamento múltiplo: é constituído por um `channel` e uma lista de `item`;
- o elemento `item` e o elemento `channel` são tuplos;
- os elementos `admin:generatorAgent` e `rdf:li` são elementos vazios;

O PFS pode ser invocado em modo *shell*, o que permite analisar ramos de uma árvore XML bem como atributos de determinados elementos.

```
[ambs@eremita XML-DT]$ pfs -shell index.rss
? item

item        =>  tup(link, dc:creator, dc:subject, title, description,
                    dc:date) * rdf:about
dc:creator  =>  text
link        =>  text
dc:subject  =>  text
title       =>  text
dc:date     =>  text
description =>  text

? li[@rdf:resource]

URL
```

```
http://null.perl-hackers.net/archives/2004/11/rsser.html
http://null.perl-hackers.net/archives/2004/11/palm_zire_31.html
http://null.perl-hackers.net/archives/2004/11/the_perl_review.html
```

Neste exemplo, o primeiro *query* permite analisar apenas a estrutura do elemento `item` (assim como os elementos necessários para a sua definição). O segundo exemplo, com uma sintaxe semelhante a XPath para referir um atributo de determinado elemento, permite inferir o tipo de dados (neste caso, URL) e mostrar o domínio activo deste atributo.

Da mesma forma que é possível obter um resumo de um documento, o PFS pode ser usado para a criação de um esqueleto de um processador em XML::DT ou para escrever um esqueleto de um DTD:

```
[ambs@eremita XML-DT]$ pfs -dt index.rss > rss.pl

[ambs@eremita XML-DT]$ pfs -dtd index.rss > rss.dtd
```

3.2 Exemplo Real

Uma outra aplicação do `pfs` é a análise reversa de documentos `HTML`. O processamento de páginas como a do Jornal Público difere muito do processamento de documentos bem formados `HTML 4.1` e muito mais do de documentos `XHTML`.

Para facilitar esta tarefa, o `pfs` pode ser usado para extrair a estrutura destes documentos (mesmo `HTML`). O exemplo seguinte foi obtido processando a página do Jornal Público online:

```
# html ...Mon Jan 24 17:33:10 2005
html    =>  tup(body, head)
body    =>  mtup(br, img, script, table+) * link * alink * vlink * topmargin
head    =>  mtup(link, meta+, script, title)
script  =>  text * language * src
br      =>  empty
table   =>  mtup(form?, tr*) * width * align * valign * height * border *
            bgcolor * cellspacing * cellpadding * id * class
img     =>  empty * width * vspace * alt * align * src * height * border *
            hspace
link    =>  empty * rel * href * type
title   =>  text
meta    =>  empty * http-equiv * content
tr      =>  mixed(td) * bgcolor * align * valign * class * height
form    =>  mtup(#PCDATA?, input?, tbody?, tr*) * action * name * id * method
td      =>  mixed(script, div, a, input, br, table, img, marquee, iframe, p,
            b, span, select) * width * onmouseover * align * valign * style *
            onmouseout * background * height * bgcolor * rowspan * colspan *
            class
tbody   =>  seq(tr)
input   =>  empty * value * name * onclick * class * type * id
div     =>  mtup(#PCDATA?, img?, script?) * align * valign
```

```
a       =>  mixed(br, strong, img, span) * target * href * onclick * title *
            class
marquee       =>  tup(p, #PCDATA) * direction * width * scrollamount * behavior *
            loop
iframe  =>  mixed(a, table, img) * width * scrolling * src * name * style *
            height * marginwidth * frameborder * id * marginheight
p       =>  mixed(br, a, strong, b, span) * align
b       =>  mixed(strong, b)
span    =>  mixed(span) * class
select  =>  seq(option) * onchange * name
strong  =>  text
option  =>  text * value
```

Da mesma forma, o `pfs` permite-nos consultar a estrutura do documento interactivamente,

```
?  name

input(name):_id  = {Submit32, Submit22, respostas, pwd, rurl, Submit}
form(name):_id   = {seccao, frmVotacao, frmAction}
select(name):_id = {seccao, dia}
iframe(name):_id = {framePlus, barraplus}
```

3.3 Estrutura interna

A ferramente PFS internamente é um programa escrito em Perl e XML::DT que processa documentos XML, e HTML e constrói uma representação interna da meta-informação dos documentos.

Durante a análise dos documentos exemplo é calculado:

- estatísticas diversas
- o conjunto de possíveis raízes (quando são vários documentos)
- para cada elemento:
  - a lista de todos os seus atributos e o respectivo tipo (exemplo: int, url, email, id, str...) e domínio activo
  - o padrão de ocorrência dos filhos: o conjunto de listas de filhos que foram encontrados
- umas ordenações breath-first (para escolher uma ordem mais didáctica de mostrar resultados

Com o cruzamento das várias tabelas são em seguida criados vários resumos que são usados para geração do DTDs, da visão resumida mostrada nos exemplos acima e para os diálogos do modo *shell*

4 Conclusões

A análise da estrutura de um documento XML é complicada. Mesmo quando se tem um DTD ou Schema, se não se possuir uma ferramenta que esquematize

gráficamente a estrutura do documento, a análise manual é morosa e sujeita a erros.

A sintaxe usada pelo PFS é clara, minimalista e os resultados apresentados ordenados topologicamente, o que permite facilidade na sua interpretação.

Por sua vez, o PFS pode ser usado como ferramenta de *bootstrap* para a definição de um DTD (no caso de o ter definido de forma *bottom-up*, criando primeiro o documento XML) ou para a criação de um esqueleto de um processador desse tipo de documentos XML usando o XML::DT.

Referências

1. J.J. Almeida and José Carlos Ramalho. XML::DT a perl down-translation module. In *XML-Europe'99, Granada - Espanha*, May 1999.
2. James Clark. Trang – multi-format schema converter based on RELAX NG. Technical report, Thai Open Source Software Center Ltd, 2003. `http://www.thaiopensource.com/relaxng/trang.html`.
3. RSS-DEV Working Group. RDF Site Summary (RSS) 1.0. Technical report, W3C - World Wide Web Consortium, 2001. `http://web.resource.org/rss/1.0/spec`.
4. Alberto Simões. XML::DT - down translating xml. *The Perl Review*, 1(1), Winter 2004.
5. W3C HTML Working Group. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). Technical report, W3C - World Wide Web Consortium, 2002. `http://www.w3.org/TR/xhtml1/`.