

# Xml\*

Christian Sternagel and René Thiemann

February 23, 2021

## Abstract

This entry provides an “XML library” for Isabelle/HOL. This includes parsing and pretty printing of XML trees as well as combinators for transforming XML trees into arbitrary user-defined data. The main contribution of this entry is an interface (fit for code generation) that allows for communication between verified programs formalized in Isabelle/HOL and the outside world via XML. This library was developed as part of the IsaFoR/CeTA project to which we refer for examples of its usage.

## Contents

<b>1</b>	<b>Parsing and Printing XML Documents</b>	<b>1</b>
1.1	Printing of XML Nodes and Documents . . . . .	2
1.2	XML-Parsing . . . . .	3
<b>2</b>	<b>XML Transformers for Extracting Data from XML Nodes</b>	<b>14</b>

## 1 Parsing and Printing XML Documents

```
theory Xml
imports
  Certification-Monads.Parser-Monad
  HOL-Library.Char-ord
begin

datatype xml =
  — node-name, attributes, child-nodes
  XML string (string × string) list xml list |
  XML-text string
```

```
datatype xmldoc =
```

---

\*This research is supported by FWF (Austrian Science Fund) projects J3202 and P22767.

```

— header, body
XMLDOC string list (root-node: xml)

fun tag :: xml ⇒ string where
  tag (XML name - -) = name |
  tag (XML-text -) = []
hide-const (open) tag

fun children :: xml ⇒ xml list where
  children (XML - - cs) = cs |
  children (XML-text -) = []
hide-const (open) children

fun num-children :: xml ⇒ nat where
  num-children (XML - - cs) = length cs |
  num-children (XML-text -) = 0
hide-const (open) num-children

```

## 1.1 Printing of XML Nodes and Documents

```

instantiation xml :: show
begin

definition shows-attr :: string × string ⇒ shows
where
  shows-attr av = shows (fst av) o shows-string ("=" @ snd av @ "'")

definition shows-attrs :: (string × string) list ⇒ shows
where
  shows-attrs as = foldr (λa. " " +#+ shows-attr a) as

fun shows-XML-indent :: string ⇒ nat ⇒ xml ⇒ shows
where
  shows-XML-indent ind i (XML n a c) =
    ("[" +#+ ind +#+ "<" +#+ shows n +#+ shows-attrs a +#+
    (if c = [] then shows-string "/>"
    else (
      ">" +#+
      foldr (shows-XML-indent (replicate i (CHR " ") @ ind) i) c +#+
      "]" +#+ ind +#+
      "</" +#+ shows n +#+ shows-string ">")) |
  shows-XML-indent ind i (XML-text t) = shows-string t

definition shows-prec (d::nat) xml = shows-XML-indent "" 2 xml

definition shows-list (xs :: xml list) = showsp-list shows-prec 0 xs

lemma shows-attr-append:
  (s +#+ shows-attr av) (r @ t) = (s +#+ shows-attr av) r @ t

```

```

unfolding shows-attr-def by (cases av) (auto simp: show-law-simps)

lemma shows-attrs-append [show-law-simps]:
  shows-attrs as (r @ s) = shows-attrs as r @ s
using shows-attr-append by (induct as) (simp-all add: shows-attrs-def)

lemma append-xml':
  shows-XML-indent ind i xml (r @ s) = shows-XML-indent ind i xml r @ s
by (induct xml arbitrary: ind r s) (auto simp: show-law-simps)

lemma shows-prec-xml-append [show-law-simps]:
  shows-prec d (xml::xml) (r @ s) = shows-prec d xml r @ s
unfolding shows-prec-xml-def by (rule append-xml')

instance
  by standard (simp-all add: show-law-simps shows-list-xml-def)

end

instantiation xmldoc :: show
begin

fun shows-xmldoc
where
  shows-xmldoc (XMLDOC h x) = shows-lines h o shows-nl o shows x

definition shows-prec (d::nat) doc = shows-xmldoc doc
definition shows-list (xs :: xmldoc list) = showsp-list shows-prec 0 xs

lemma shows-prec-xmldoc-append [show-law-simps]:
  shows-prec d (x::xmldoc) (r @ s) = shows-prec d x r @ s
by (cases x) (auto simp: shows-prec-xmldoc-def show-law-simps)

instance
  by standard (simp-all add: show-law-simps shows-list-xmldoc-def)

end

```

## 1.2 XML-Parsing

```

definition parse-text :: string option parser
where
  parse-text = do {
    ts ← many ((≠) CHR "<");
    let text = trim ts;
    if text = [] then return None
    else return (Some (List.rev (trim (List.rev text))))
  }

```

**lemma** *is-parser-parse-text* [intro]:

*is-parser parse-text*

**by** (*auto simp: parse-text-def*)

**lemma** *parse-text-consumes*:

**assumes** \*:  $ts \neq []$   $hd\ ts \neq CHR\ "<"$

**and** *parse*: *parse-text*  $ts = Inr\ (t, ts')$

**shows**  $length\ ts' < length\ ts$

**proof** –

**from** \* **obtain**  $a\ tss$  **where**  $ts: ts = a \# tss$  **and**  $not: a \neq CHR\ "<"$

**by** (*cases*  $ts$ , *auto*)

**note** *parse* = *parse* [*unfolded parse-text-def Let-def*  $ts$ ]

**from** *parse* **obtain**  $x1\ x2$  **where** *many*:  $many\ ((\neq)\ CHR\ "<")\ tss = Inr\ (x1,$   
 $x2)$

**using** *not* **by** (*cases*  $many\ ((\neq)\ CHR\ "<")\ tss$ ,

*auto simp: bind-def*)

**from** *is-parser-many* *many* **have**  $len: length\ x2 \leq length\ tss$  **by** *blast*

**from** *parse* *many* **have**  $length\ ts' \leq length\ x2$

**using** *not* **by** (*simp add: bind-def return-def split: if-splits*)

**with**  $len$  **show** *?thesis* **unfolding**  $ts$  **by** *auto*

**qed**

**definition** *parse-attribute-value* :: *string parser*

**where**

*parse-attribute-value* = *do* {

*exactly* [ $CHR\ "'";$

$v \leftarrow many\ ((\neq)\ CHR\ "'");$

*exactly* [ $CHR\ "'";$

*return*  $v$

}

**lemma** *is-parser-parse-attribute-value* [intro]:

*is-parser parse-attribute-value*

**by** (*auto simp: parse-attribute-value-def*)

A list of characters that are considered to be "letters" for tag-names.

**definition** *letters* :: *char list*

**where**

$letters = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-0123456789&,:;-"$

**definition** *is-letter* :: *char*  $\Rightarrow$  *bool*

**where**

*is-letter*  $c \longleftrightarrow c \in set\ letters$

**lemma** *is-letter-code* [code]:

*is-letter*  $c \longleftrightarrow$

$CHR\ "a" \leq c \wedge c \leq CHR\ "z" \vee$

$CHR\ "A" \leq c \wedge c \leq CHR\ "Z" \vee$

$CHR\ "0" \leq c \wedge c \leq CHR\ "9" \vee$

$c \in \text{set } \text{"\&::-"}'$   
**by** (cases c) (simp add: is-letter-def letters-def)

**definition** many-letters :: string parser  
**where**

[simp]: many-letters = manyof letters

**lemma** many-letters [code, code-unfold]:

many-letters = many is-letter

**by** (simp add: is-letter-def [abs-def] manyof-def)

**definition** parse-name :: string parser

**where**

parse-name s = (do {  
 n ← many-letters;  
 spaces;  
 if n = [] then  
 error ("expected letter " @ letters @ " but first symbol is " @ take 1 s @ """)  
 else return n  
}) s

**lemma** is-parser-parse-name [intro]:

is-parser parse-name

**proof**

**fix** s r x

**assume** res: parse-name s = Inr (x, r)

**let** ?exp = do {

n ← many-letters;

spaces;

if n = [] then

error ("expected letter " @ letters @ " but first symbol is " @ take 1 s @ """)

else return n

}

**have** isp: is-parser ?exp **by** auto

**have** id: parse-name s = ?exp s **by** (simp add: parse-name-def)

**from** isp [unfolded is-parser-def, rule-format, OF res [unfolded id]]

**show** length r ≤ length s .

**qed**

**function** (sequential) parse-attributes :: (string × string) list parser

**where**

parse-attributes [] = Error-Monad.return ([], []) |

parse-attributes (c # s) =

(if c ∈ set "/>" then Error-Monad.return ([], c # s)

else (do {

k ← parse-name;

exactly "=";

v ← parse-attribute-value;

atts ← parse-attributes;

```

    return ((k, v) # atts)
  }) (c # s)
  by pat-completeness auto

```

**termination** *parse-attributes*

**proof**

```

  show wf (measure length) by simp

```

**next**

```

  fix c s y ts ya tsa yb tsb

```

```

  assume pn: parse-name (c # s) = Inr (y, ts)

```

```

    and oo: exactly "=" ts = Inr (ya, tsa)

```

```

    and pav: parse-attribute-value tsa = Inr (yb, tsb)

```

```

  have cp: is-cparser (exactly "=") by auto

```

```

  from cp [unfolded is-cparser-def] oo have 1: length ts > length tsa by auto

```

```

  from is-parser-parse-name [unfolded is-parser-def] pn

```

```

    have 2: length (c # s) ≥ length ts by force

```

```

  from is-parser-parse-attribute-value [unfolded is-parser-def] pav

```

```

    have 3: length tsa ≥ length tsb by force

```

```

  from 1 2 3

```

```

    show (tsb, c # s) ∈ measure length

```

```

  by auto

```

**qed**

**lemma** *is-parser-parse-attributes* [intro]:

*is-parser parse-attributes*

**proof**

```

  fix s r x

```

```

  assume parse-attributes s = Inr (x, r)

```

```

  then show length r ≤ length s

```

```

  proof (induct arbitrary: x rule: parse-attributes.induct)

```

```

    case (2 c s)

```

```

    show ?case

```

```

    proof (cases c ∈ set ">")

```

```

      case True

```

```

        with 2(2) show ?thesis by simp

```

```

    next

```

```

      case False

```

```

        from False 2(2) obtain y1 s1

```

```

          where pn: parse-name (c # s) = Inr (y1, s1)

```

```

          by (cases parse-name (c # s)) (auto simp: bind-def)

```

```

        from False 2(2) pn obtain y2 s2

```

```

          where oo: exactly "=" s1 = Inr (y2, s2)

```

```

          by (cases exactly "=" s1) (auto simp: bind-def)

```

```

        from False 2(2) pn oo obtain y3 s3

```

```

          where pav: parse-attribute-value s2 = Inr (y3, s3)

```

```

          by (cases parse-attribute-value s2) (auto simp: bind-def)

```

```

        from False 2(2) pn oo pav obtain y4

```

```

          where patts: parse-attributes s3 = Inr (y4, r)

```

```

          by (cases parse-attributes s3) (auto simp: return-def bind-def)

```

```

    have length r ≤ length s3 using 2(1)[OF False pn oo pav patts] .
    also have ... ≤ length s2
      using is-parser-parse-attribute-value [unfolded is-parser-def] pav by auto
    also have ... ≤ length s1 using is-parser-exactly [unfolded is-parser-def] oo
  by auto
    also have ... ≤ length (c # s)
      using is-parser-parse-name [unfolded is-parser-def] pn by force
    finally show length r ≤ length (c # s) by auto
  qed
  qed simp
  qed

```

```

context notes [[function-internals]]
begin

```

```

function parse-nodes :: xml list parser
where

```

```

  parse-nodes ts =
    (if ts = [] ∨ take 2 ts = "</" then return [] ts
     else if hd ts ≠ CHR "<" then (do {
       t ← parse-text;
       ns ← parse-nodes;
       return (XML-text (the t) # ns)
     }) ts
     else (do {
       exactly "<";
       n ← parse-name;
       atts ← parse-attributes;
       e ← oneof ["/>", ">"];
       (λ ts'.
         if e = "/>" then (do {
           cs ← parse-nodes;
           return (XML n atts [] # cs)
         }) ts' else (do {
           cs ← parse-nodes;
           exactly "<";
           exactly n;
           exactly ">";
           ns ← parse-nodes;
           return (XML n atts cs # ns)
         }) ts')
     }) ts)
  by pat-completeness auto

```

```

end

```

```

lemma parse-nodes-help:

```

```

  parse-nodes-dom s ∧ (∀ x r. parse-nodes s = Inr (x, r) → length r ≤ length s)
(is ?prop s)

```

```

proof (induct rule: wf-induct [where P = ?prop and r = measure length])
  fix s
  assume  $\forall t. (t, s) \in \text{measure length} \longrightarrow ?\text{prop } t$ 
  then have ind1:  $\bigwedge t. \text{length } t < \text{length } s \implies \text{parse-nodes-dom } t$ 
    and ind2:  $\bigwedge t \ x \ r. \text{length } t < \text{length } s \implies \text{parse-nodes } t = \text{Inr } (x,r) \implies \text{length}$ 
r  $\leq \text{length } t$  by auto
  let ?check =  $\lambda s. s = [] \vee \text{take } 2 \ s = "</"$ 
  let ?check2 =  $\text{hd } s \neq \text{CHR } "<"$ 
  have dom: parse-nodes-dom s
  proof
    fix y
    assume parse-nodes-rel y s
    then show parse-nodes-dom y
    proof
      fix ts ya tsa
      assume *:  $y = \text{tsa } s = \text{ts} \wedge (ts = [] \vee \text{take } 2 \ ts = "</")$ 
         $\text{hd } ts \neq \text{CHR } "<"$  and parse: parse-text ts = Inr (ya, tsa)
      from parse-text-consumes[OF - - parse] *(3-4) have length tsa < length ts
    by auto
    with * have len: length s > length y by simp
    from ind1[OF this] show parse-nodes-dom y .
  next
    fix ts ya tsa yaa tsb yb tsc yc tsd
    assume y = tsd and s = ts and  $\neg ?\text{check } ts$ 
      and exactly "<" ts = Inr (ya, tsa)
      and parse-name tsa = Inr (yaa, tsb)
      and parse-attributes tsb = Inr (yb, tsc)
      and oneof ["/>", ">"] tsc = Inr (yc, tsd)
      and yc = "/>"
    then have len: length s > length y
      using is-cparser-exactly [of "<"]
      and is-parser-oneof [of ["/>", ">"]]
      and is-parser-parse-attributes
      and is-parser-parse-name
      by (auto dest!: is-parser-length is-cparser-length)
    with ind1[OF len] show parse-nodes-dom y by simp
  next
    fix ts ya tsa yaa tsb yb tsc yc tsd
    assume y = tsd and s = ts and  $\neg ?\text{check } ts$ 
      and exactly "<" ts = Inr (ya, tsa)
      and parse-name tsa = Inr (yaa, tsb)
      and parse-attributes tsb = Inr (yb, tsc)
      and oneof ["/>", ">"] tsc = Inr (yc, tsd)
    then have len: length s > length y
      using is-cparser-exactly [of "<", simplified]
      and is-parser-oneof [of ["/>", ">"]]
      and is-parser-parse-attributes
      and is-parser-parse-name
      by (auto dest!: is-parser-length is-cparser-length)

```



```

with ind1[OF len] show parse-nodes-dom y by simp
next
fix ts ya tsa yaa tsb yb tsc yc tse ye tsf yf tsg yg tsh yh tsi yi tsj
assume y: y = tsj and s = ts and ¬ ?check ts
  and exactly "<" ts = Inr (ya, tsa)
  and parse-name tsa = Inr (yaa, tsb)
  and parse-attributes tsb = Inr (yb, tsc)
  and oneof ["/>", ">"] tsc = Inr (yc, tse)
  and rec: parse-nodes-sumC tse = Inr (ye, tsf)
  and last: exactly "</" tsf = Inr (yf, tsg)
  exactly yaa tsg = Inr (yg, tsh)
  exactly ">" tsh = Inr (yi, tsj)
then have len: length s > length tse
  using is-cparser-exactly [of "<", simplified]
  and is-parser-oneof [of ["/>", ">"]]
  and is-parser-parse-attributes
  and is-parser-parse-name
  by (auto dest!: is-parser-length is-cparser-length)
from last(1) last(2) have len2a: length tsf ≥ length tsh
  using is-parser-exactly [of "</"] and is-parser-exactly [of yaa]
  and is-parser-parse-name by (auto dest!: is-parser-length)
have len2c: length tsh ≥ length y using last(3)
  using is-parser-exactly [of ">"] by (auto simp: y dest!: is-parser-length)
from len2a len2c have len2: length tsf ≥ length y by simp
from ind2[OF len rec[unfolded parse-nodes-def[symmetric]]] len len2 have
length s > length y by simp
from ind1[OF this]
  show parse-nodes-dom y .
qed
qed
note psimps = parse-nodes.psimps[OF dom]
show ?prop s
proof (intro conjI, rule dom, intro allI impI)
  fix x r
  assume res: parse-nodes s = Inr (x,r)
  note res = res[unfolded psimps]
  then show length r ≤ length s
  proof (cases ?check s)
    case True
    then show ?thesis using res by (simp add: return-def)
  next
  case False note oFalse = this
  show ?thesis
  proof (cases ?check2)
    case True
    note res = res[simplified False True, simplified]
    from res obtain y1 s1 where pt: parse-text s = Inr (y1, s1) by (cases
parse-text s, auto simp: bind-def)
    note res = res[unfolded bind-def pt, simplified]

```

```

from res obtain y2 s2
  where pn: parse-nodes s1 = Inr (y2, s2)
  by (cases parse-nodes s1) (auto simp: bind-def)
note res = res[simplified bind-def pn, simplified]
from res have r: r = s2 by (simp add: return-def bind-def)
from parse-text-consumes[OF - True pt] False
have lens: length s1 < length s by auto
from ind2[OF lens pn] have length s2 ≤ length s1 .
then show ?thesis using lens unfolding r by auto
next
case False note ooFalse = this
note res = res[simplified ooFalse ooFalse, simplified]
from res obtain y1 s1 where oo: exactly "<" s = Inr (y1, s1) by (cases
exactly "<" s, auto simp: bind-def)
note res = res[unfolded bind-def oo, simplified]
from res obtain y2 s2
  where pn: parse-name s1 = Inr (y2, s2)
  by (cases parse-name s1) (auto simp: bind-def psimps)
note res = res[simplified bind-def pn, simplified]
from res obtain y3 s3 where pa: parse-attributes s2 = Inr (y3, s3)
  by (cases parse-attributes s2) (auto simp: return-def bind-def)
note res = res[simplified pa, simplified]
from res obtain y4 s4
  where oo2: oneof ["/>", ">"] s3 = Inr (y4, s4)
  by (cases oneof ["/>", ">"] s3) (auto simp: return-def bind-def)
note res = res[unfolded oo2, simplified]
from is-parser-parse-attributes and is-parser-oneof [of ["/>", ">"]]
  and is-cparser-exactly [of "<", simplified] and is-parser-parse-name
  and oo pn pa oo2
  have s-s4: length s > length s4
  by (auto dest!: is-parser-length is-cparser-length)
show ?thesis
proof (cases y4 = "/>")
  case True
    from res True obtain y5
      where pns: parse-nodes s4 = Inr (y5, r)
      by (cases parse-nodes s4) (auto simp: return-def bind-def)
      from ind2[OF s-s4 pns] s-s4 show length r ≤ length s by simp
    next
      case False
        note res = res[simplified False, simplified]
        from res obtain y6 s6 where pns: parse-nodes s4 = Inr (y6, s6)
          by (cases parse-nodes s4) (auto simp: return-def bind-def)
        note res = res[unfolded bind-def pns, simplified, unfolded bind-def]
        from res obtain y7 s7 where oo3: exactly "</" s6 = Inr (y7, s7) by
(cases exactly "</" s6, auto)
        note res = res[unfolded oo3, simplified, unfolded bind-def,
simplified, unfolded bind-def]
        from res obtain y8 s8 where oo4: exactly y2 s7 = Inr (y8, s8) by (cases

```

```

exactly y2 s7, auto)
  note res = res[unfolded oo4 bind-def, simplified]
  from res obtain y10 s10 where oo5: exactly ">" s8 = Inr (y10,s10)
  by (cases exactly ">" s8, auto simp: bind-def)
  note res = res[unfolded oo5 bind-def, simplified]
  from res obtain y11 s11 where pns2: parse-nodes s10 = Inr (y11, s11)
by (cases parse-nodes s10, auto simp: bind-def)
  note res = res[unfolded bind-def pns2, simplified]
  note one = is-parser-oneof [unfolded is-parser-def, rule-format]
  note exact = is-parser-exactly [unfolded is-parser-def, rule-format]
  from ind2[OF s-s4 pns] s-s4 exact[OF oo3] exact[OF oo4]
  have s-s7: length s > length s8 unfolding is-parser-def by force
  with exact[OF oo5] have s-s10: length s > length s10 by simp
  with ind2[OF s-s10 pns2] have s-s11: length s > length s11 by simp
  then show length r ≤ length s using res by (auto simp: return-def)
  qed
  qed
  qed
  qed
qed simp

```

**termination** *parse-nodes* **using** *parse-nodes-help* **by** *blast*

**lemma** *parse-nodes* [*intro*]:  
*is-parser parse-nodes*  
**unfolding** *is-parser-def* **using** *parse-nodes-help* **by** *blast*

A more efficient variant of *oneof* [*"/>", ">"*].

```

fun oneof-closed :: string parser
where
  oneof-closed (x # xs) =
    (if x = CHR ">" then Error-Monad.return (">", trim xs)
     else if x = CHR "/" ∧ (case xs of [] ⇒ False | y # ys ⇒ y = CHR ">") then
       Error-Monad.return ("/>", trim (tl xs))
     else err-expecting ("one of [/>, >]" (x # xs)) |
  oneof-closed xs = err-expecting ("one of [/>, >]" xs

```

```

lemma oneof-closed:
  oneof ["/>", ">"] = oneof-closed (is ?l = ?r)
proof (rule ext)
  fix xs
  have id: "one of " @ shows-list ["/>", ">"] [] = "one of [/>, >]"
  by (simp add: shows-list-list-def showsp-list-def pshowsp-list-def shows-list-gen-def
      shows-string-def shows-prec-list-def shows-list-char-def)
  note d = oneof-def oneof-aux.simps id
  show ?l xs = ?r xs
  proof (cases xs)
  case Nil
  show ?thesis unfolding Nil d by simp

```

```

next
  case (Cons x xs) note oCons = this
  show ?thesis
  proof (cases x = CHR ">")
    case True
    show ?thesis unfolding Cons d True by simp
  next
  case False note oFalse = this
  show ?thesis
  proof (cases x = CHR "/"')
    case False
    show ?thesis unfolding Cons d using False oFalse by simp
  next
  case True
  show ?thesis
  proof (cases xs)
    case Nil
    show ?thesis unfolding Cons Nil d by auto
  next
  case (Cons y ys)
  show ?thesis unfolding oCons Cons d by simp
  qed
qed
qed
qed
qed

```

**lemma** *If-removal*:

$(\lambda e x. \text{if } b \ e \ \text{then } f \ e \ x \ \text{else } g \ e \ x) = (\lambda e. \text{if } b \ e \ \text{then } f \ e \ \text{else } g \ e)$   
**by** (*intro ext*) *auto*

**declare** *parse-nodes.simps* [*unfolded oneof-closed*,  
*unfolded If-removal* [*of*  $\lambda e. e = \text{">"}$ ], *code*]

**definition** *parse-node* :: *xml parser*

**where**

```

parse-node = do {
  exactly "<";
  n ← parse-name;
  atts ← parse-attributes;
  e ← oneof [">", ">'"];
  if e = ">" then return (XML n atts [])
  else do {
    cs ← parse-nodes;
    exactly "</";
    exactly n;
    exactly ">";
    return (XML n atts cs)
  }
}

```

```

}

declare parse-node-def [unfolded oneof-closed, code]

function parse-header :: string list parser
where
  parse-header ts =
    (if take 2 (trim ts) = "<?" then (do {
      h ← scan-upto "?>";
      hs ← parse-header;
      return (h # hs)
    }) ts else (do {
      spaces;
      return []
    }) ts)
by pat-completeness auto

termination parse-header
proof
  fix ts y tsa
  assume scan-upto "?>" ts = Inr (y, tsa)
  with is-cparser-scan-upto have length ts > length tsa
  unfolding is-cparser-def by force
  then show (tsa, ts) ∈ measure length by simp
qed simp

consts scala :: bool

code-printing
constant scala ↪
  (Haskell) False and
  (Scala) true and
  (SML) false and
  (OCaml) false and
  (Eval) false

fun remove-comments-aux :: bool ⇒ string ⇒ string
where
  remove-comments-aux False (c # cs) =
    (if c = CHR "<" ∧ take 3 cs = "!--" then remove-comments-aux True (tl cs)
     else c # remove-comments-aux False cs) |
  remove-comments-aux True (c # cs) =
    (if c = CHR "-" ∧ take 2 cs = "->" then remove-comments-aux False (drop
  2 cs)
     else remove-comments-aux True cs) |
  remove-comments-aux - [] = []

fun remove-comments-aux-acc :: string ⇒ bool ⇒ string ⇒ string
where

```

```

remove-comments-aux-acc a False (c # cs) =
  (if c = CHR "<" ∧ take 3 cs = "!--" then remove-comments-aux-acc a True
(tl cs)
  else remove-comments-aux-acc (c # a) False cs) |
remove-comments-aux-acc a True (c # cs) =
  (if c = CHR "-" ∧ take 2 cs = "->" then remove-comments-aux-acc a False
(drop 2 cs)
  else remove-comments-aux-acc a True cs) |
remove-comments-aux-acc a [] = a

```

A tail recursive variant for Scala to reduce stack size at the cost of double traversal.

```

definition remove-comments :: string ⇒ string
where
  remove-comments =
    (if scala then rev o (remove-comments-aux-acc [] False)
    else remove-comments-aux False)

```

```

hide-const remove-comments-aux remove-comments-aux-acc

```

```

definition parse-doc :: xmldoc parser
where
  parse-doc = do {
    update-tokens remove-comments;
    h ← parse-header;
    xml ← parse-node;
    eoi;
    return (XMLDOC h xml)
  }

```

```

definition doc-of-string :: string ⇒ string + xmldoc
where
  doc-of-string s = do {
    (doc, -) ← parse-doc s;
    Error-Monad.return doc
  }

```

```

end

```

## 2 XML Transformers for Extracting Data from XML Nodes

```

theory Xmlt
imports
  Xml
  Certification-Monads.Strict-Sum
  HOL.Rat
begin

```

**type-synonym**

*tag = string*

The type of transformers on xml nodes.

**type-synonym**

*'a xmlt = xml  $\Rightarrow$  string  $+_{\perp}$  'a*

**definition** *map :: (xml  $\Rightarrow$  ('e  $+_{\perp}$  'a))  $\Rightarrow$  xml list  $\Rightarrow$  'e  $+_{\perp}$  'a list*

**where**

*[code-unfold]: map = map-sum-bot*

**lemma** *map-mono [partial-function-mono]:*

**fixes** *C :: xml  $\Rightarrow$  ('b  $\Rightarrow$  ('e  $+_{\perp}$  'c))  $\Rightarrow$  'e  $+_{\perp}$  'd*

**assumes** *C:  $\bigwedge y. y \in \text{set } B \implies \text{mono-sum-bot } (C y)$*

**shows** *mono-sum-bot ( $\lambda f. \text{map } (\lambda y. C y f) B$ )*

**unfolding** *map-def by (auto intro: partial-function-mono C)*

**hide-const (open) map**

**fun** *text :: tag  $\Rightarrow$  string xmlt*

**where**

*text tag (XML n atts [XML-text t]) =*

*(if n = tag  $\wedge$  atts = [] then return t*

*else error (concat*

*["could not extract text for ", tag, " from ", " $\boxed{\leftarrow}$ ", show (XML n atts [XML-text t])]))*

*| text tag xml = error (concat ["could not extract text for ", tag, " from ", " $\boxed{\leftarrow}$ ", show xml])*

**hide-const (open) text**

**definition** *bool-of-string :: string  $\Rightarrow$  string  $+_{\perp}$  bool*

**where**

*bool-of-string s =*

*(if s = "true" then return True*

*else if s = "false" then return False*

*else error ("cannot convert " @ s @ " into Boolean"))*

**fun** *bool :: tag  $\Rightarrow$  bool xmlt*

**where**

*bool tag node = Xmlt.text tag node  $\gg=$  bool-of-string*

**hide-const (open) bool**

**definition** *fail :: tag  $\Rightarrow$  'a xmlt*

**where**

*fail tag xml =*

*error (concat*

*["could not transform the following xml element (expected ", tag, ") ", " $\boxed{\leftarrow}$ ", show xml])*

**hide-const** (open) fail

**definition** guard :: (xml  $\Rightarrow$  bool)  $\Rightarrow$  'a xmlt  $\Rightarrow$  'a xmlt  $\Rightarrow$  'a xmlt  
**where**

guard p p1 p2 x = (if p x then p1 x else p2 x)

**hide-const** (open) guard

**lemma** guard-mono [partial-function-mono]:

**assumes** p1:  $\bigwedge y. \text{mono-sum-bot } (p1 \ y)$

**and** p2:  $\bigwedge y. \text{mono-sum-bot } (p2 \ y)$

**shows** mono-sum-bot ( $\lambda g. \text{Xmlt.guard } p \ (\lambda y. p1 \ y \ g) \ (\lambda y. p2 \ y \ g) \ x$ )

**by** (cases x) (auto intro!: partial-function-mono p1 p2 simp: Xmlt.guard-def)

**fun** leaf :: tag  $\Rightarrow$  'a  $\Rightarrow$  'a xmlt

**where**

leaf tag x (XML name atts cs) =  
(if name = tag  $\wedge$  atts = []  $\wedge$  cs = [] then return x  
else Xmlt.fail tag (XML name atts cs)) |

leaf tag x xml = Xmlt.fail tag xml

**hide-const** (open) leaf

**fun** list1element :: 'a list  $\Rightarrow$  'a option

**where**

list1element [x] = Some x |

list1element - = None

**fun** singleton :: tag  $\Rightarrow$  'a xmlt  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'b xmlt

**where**

singleton tag p1 f xml =  
(case xml of  
XML name atts cs  $\Rightarrow$   
(if name = tag  $\wedge$  atts = [] then  
(case list1element cs of  
Some (cs1)  $\Rightarrow$  p1 cs1  $\gg$  return  $\circ$  f  
| None  $\Rightarrow$  Xmlt.fail tag xml)  
else Xmlt.fail tag xml)  
| -  $\Rightarrow$  Xmlt.fail tag xml)

**hide-const** (open) singleton

**lemma** singleton-mono [partial-function-mono]:

**assumes** p:  $\bigwedge y. \text{mono-sum-bot } (p1 \ y)$

**shows** mono-sum-bot ( $\lambda g. \text{Xmlt.singleton } t \ (\lambda y. p1 \ y \ g) \ f \ x$ )

**by** (cases x, cases list1element (Xml.children x)) (auto intro!: partial-function-mono p)

**fun** list2elements :: 'a list  $\Rightarrow$  ('a  $\times$  'a) option

**where**

list2elements [x, y] = Some (x, y) |

list2elements - = None



**fun** pair :: tag ⇒ 'a xmlt ⇒ 'b xmlt ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'c xmlt

**where**

```
pair tag p1 p2 f xml =
  (case xml of
    XML name atts cs ⇒
      (if name = tag ∧ atts = [] then
        (case list2elements cs of
          Some (cs1, cs2) ⇒
            do {
              a ← p1 cs1;
              b ← p2 cs2;
              return (f a b)
            }
          | None ⇒ Xmlt.fail tag xml)
      else Xmlt.fail tag xml)
  | - ⇒ Xmlt.fail tag xml)
```

**hide-const (open) pair**

**lemma** pair-mono [partial-function-mono]:

**assumes**  $\bigwedge y. \text{mono-sum-bot } (p1\ y)$

**and**  $\bigwedge y. \text{mono-sum-bot } (p2\ y)$

**shows**  $\text{mono-sum-bot } (\lambda g. \text{Xmlt.pair } t\ (\lambda y. p1\ y\ g)\ (\lambda y. p2\ y\ g)\ f\ x)$

**using** *assms*

**by** (*cases x, cases list2elements (Xml.children x)*) (*auto intro!: partial-function-mono*)

**fun** list3elements :: 'a list ⇒ ('a × 'a × 'a) option

**where**

*list3elements* [x, y, z] = *Some* (x, y, z) |

*list3elements* - = *None*

**fun** triple :: string ⇒ 'a xmlt ⇒ 'b xmlt ⇒ 'c xmlt ⇒ ('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'd xmlt

**where**

```
triple tag p1 p2 p3 f xml = (case xml of XML name atts cs ⇒
  (if name = tag ∧ atts = [] then
    (case list3elements cs of
      Some (cs1, cs2, cs3) ⇒
        do {
          a ← p1 cs1;
          b ← p2 cs2;
          c ← p3 cs3;
          return (f a b c)
        }
      | None ⇒ Xmlt.fail tag xml)
    else Xmlt.fail tag xml)
  | - ⇒ Xmlt.fail tag xml)
```

**lemma** triple-mono [partial-function-mono]:

```

assumes  $\bigwedge y. \text{mono-sum-bot } (p1\ y)$ 
and  $\bigwedge y. \text{mono-sum-bot } (p2\ y)$ 
and  $\bigwedge y. \text{mono-sum-bot } (p3\ y)$ 
shows  $\text{mono-sum-bot } (\lambda g. \text{Xmlt.triple } t\ (\lambda y. p1\ y\ g)\ (\lambda y. p2\ y\ g)\ (\lambda y. p3\ y\ g))$ 
 $f\ x)$ 
using assms
by (cases x, cases list3elements (Xml.children x), auto intro!: partial-function-mono)

```

```

fun list4elements :: 'a list  $\Rightarrow$  ('a  $\times$  'a  $\times$  'a  $\times$  'a) option

```

```

where

```

```

list4elements [x, y, z, u] = Some (x, y, z, u) |
list4elements - = None

```

```

fun

```

```

tuple4 ::
  string  $\Rightarrow$  'a xmlt  $\Rightarrow$  'b xmlt  $\Rightarrow$  'c xmlt  $\Rightarrow$  'd xmlt  $\Rightarrow$  ('a  $\Rightarrow$  'b  $\Rightarrow$  'c  $\Rightarrow$  'd  $\Rightarrow$  'e)
 $\Rightarrow$  'e xmlt

```

```

where

```

```

tuple4 tag p1 p2 p3 p4 f xml =
  (case xml of
    XML name atts cs  $\Rightarrow$ 
      (if name = tag  $\wedge$  atts = [] then
        (case list4elements cs of
          Some (cs1, cs2, cs3, cs4)  $\Rightarrow$ 
            do {
              a  $\leftarrow$  p1 cs1;
              b  $\leftarrow$  p2 cs2;
              c  $\leftarrow$  p3 cs3;
              d  $\leftarrow$  p4 cs4;
              return (f a b c d)
            }
          | None  $\Rightarrow$  Xmlt.fail tag xml)
        else Xmlt.fail tag xml)
    | -  $\Rightarrow$  Xmlt.fail tag xml)

```

```

lemma tuple4-mono [partial-function-mono]:

```

```

assumes  $\bigwedge y. \text{mono-sum-bot } (p1\ y)$ 
and  $\bigwedge y. \text{mono-sum-bot } (p2\ y)$ 
and  $\bigwedge y. \text{mono-sum-bot } (p3\ y)$ 
and  $\bigwedge y. \text{mono-sum-bot } (p4\ y)$ 
shows  $\text{mono-sum-bot } (\lambda g. \text{Xmlt.tuple4 } t\ (\lambda y. p1\ y\ g)\ (\lambda y. p2\ y\ g)\ (\lambda y. p3\ y\ g)$ 
 $(\lambda y. p4\ y\ g))\ f\ x)$ 
using assms
by (cases x, cases list4elements (Xml.children x) (auto intro!: partial-function-mono))

```

```

fun list5elements :: 'a list  $\Rightarrow$  ('a  $\times$  'a  $\times$  'a  $\times$  'a  $\times$  'a) option

```

```

where

```

```

list5elements [x, y, z, u, v] = Some (x, y, z, u, v) |
list5elements - = None

```

```

fun
  tuple5 ::
    string ⇒ 'a xmlt ⇒ 'b xmlt ⇒ 'c xmlt ⇒ 'd xmlt ⇒ 'e xmlt ⇒
      ('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e ⇒ 'f) ⇒ 'f xmlt
where
  tuple5 tag p1 p2 p3 p4 p5 f xml =
    (case xml of
      XML name atts cs ⇒
        (if name = tag ∧ atts = [] then
          (case list5elements cs of
            Some (cs1,cs2,cs3,cs4,cs5) ⇒
              do {
                a ← p1 cs1;
                b ← p2 cs2;
                c ← p3 cs3;
                d ← p4 cs4;
                e ← p5 cs5;
                return (f a b c d e)
              }
          | None ⇒ Xmlt.fail tag xml)
        else Xmlt.fail tag xml)
    | - ⇒ Xmlt.fail tag xml)

lemma tuple5-mono [partial-function-mono]:
  assumes ∧y. mono-sum-bot (p1 y)
  and ∧y. mono-sum-bot (p2 y)
  and ∧y. mono-sum-bot (p3 y)
  and ∧y. mono-sum-bot (p4 y)
  and ∧y. mono-sum-bot (p5 y)
  shows mono-sum-bot (λg. Xmlt.tuple5 t (λy. p1 y g) (λ y. p2 y g) (λ y. p3 y g)
    (λ y. p4 y g) (λ y. p5 y g) f x)
  using assms
  by (cases x, cases list5elements (Xml.children x)) (auto intro!: partial-function-mono)

fun list6elements :: 'a list ⇒ ('a × 'a × 'a × 'a × 'a × 'a) option
where
  list6elements [x, y, z, u, v, w] = Some (x, y, z, u, v, w) |
  list6elements - = None

fun
  tuple6 ::
    string ⇒ 'a xmlt ⇒ 'b xmlt ⇒ 'c xmlt ⇒ 'd xmlt ⇒ 'e xmlt ⇒ 'f xmlt ⇒
      ('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'g) ⇒ 'g xmlt
where
  tuple6 tag p1 p2 p3 p4 p5 p6 f xml =
    (case xml of
      XML name atts cs ⇒
        (if name = tag ∧ atts = [] then

```

```

(case list6elements cs of
  Some (cs1,cs2,cs3,cs4,cs5,cs6) =>
    do {
      a ← p1 cs1;
      b ← p2 cs2;
      c ← p3 cs3;
      d ← p4 cs4;
      e ← p5 cs5;
      ff ← p6 cs6;
      return (f a b c d e ff)
    }
  | None => Xmlt.fail tag xml)
else Xmlt.fail tag xml)
| - => Xmlt.fail tag xml)

```

**lemma** *tuple6-mono* [*partial-function-mono*]:

```

assumes  $\bigwedge y. \text{mono-sum-bot } (p1\ y)$ 
and  $\bigwedge y. \text{mono-sum-bot } (p2\ y)$ 
and  $\bigwedge y. \text{mono-sum-bot } (p3\ y)$ 
and  $\bigwedge y. \text{mono-sum-bot } (p4\ y)$ 
and  $\bigwedge y. \text{mono-sum-bot } (p5\ y)$ 
and  $\bigwedge y. \text{mono-sum-bot } (p6\ y)$ 
shows  $\text{mono-sum-bot } (\lambda g. \text{Xmlt.tuple6 } t\ (\lambda y. p1\ y\ g)\ (\lambda y. p2\ y\ g)\ (\lambda y. p3\ y\ g)$ 
 $(\lambda y. p4\ y\ g)\ (\lambda y. p5\ y\ g)\ (\lambda y. p6\ y\ g)\ f\ x)$ 
using assms
by (cases x, cases list6elements (Xml.children x)) (auto intro!: partial-function-mono)

```

**fun** *optional* :: *tag* => '*a xmlt* => ('*a option* => '*b*) => '*b xmlt*

**where**

```

optional tag p1 f (XML name atts cs) =
  (let l = length cs in
    (if name = tag ^ atts = [] ^ l ≥ 0 ^ l ≤ 1 then do {
      if l = 1 then do {
        x1 ← p1 (cs ! 0);
        return (f (Some x1))
      } else return (f None)
    } else Xmlt.fail tag (XML name atts cs))) |
optional tag p1 f xml = Xmlt.fail tag xml

```

**lemma** *optional-mono* [*partial-function-mono*]:

```

assumes  $\bigwedge y. \text{mono-sum-bot } (p1\ y)$ 
shows  $\text{mono-sum-bot } (\lambda g. \text{Xmlt.optional } t\ (\lambda y. p1\ y\ g)\ f\ x)$ 
using assms by (cases x) (auto intro!: partial-function-mono)

```

**fun** *xml1to2elements* :: *string* => '*a xmlt* => '*b xmlt* => ('*a* => '*b option* => '*c*) => '*c xmlt*

**where**

```

xml1to2elements tag p1 p2 f (XML name atts cs) = (
  let l = length cs in

```

```

(if name = tag ∧ atts = [] ∧ l ≥ 1 ∧ l ≤ 2
 then do {
   x1 ← p1 (cs ! 0);
   (if l = 2
    then do {
      x2 ← p2 (cs ! 1);
      return (f x1 (Some x2))
    } else return (f x1 None))
 } else Xmlt.fail tag (XML name atts cs)) |
xml1to2elements tag p1 p2 f xml = Xmlt.fail tag xml

```

**lemma** *xml1to2elements-mono* [*partial-function-mono*]:

```

assumes p1:  $\bigwedge y. \text{mono-sum-bot } (p1 \ y)$ 
           $\bigwedge y. \text{mono-sum-bot } (p2 \ y)$ 
shows  $\text{mono-sum-bot } (\lambda g. \text{xml1to2elements } t \ (\lambda y. p1 \ y \ g) \ (\lambda y. p2 \ y \ g) \ f \ x)$ 
by (cases x, auto intro!: partial-function-mono p1)

```

Apply the first transformer to the first child-node, then check the second child-node, which is must be a Boolean. If the Boolean is true, then apply the second transformer to the last child-node.

```

fun xml2nd-choice :: tag ⇒ 'a xmlt ⇒ tag ⇒ 'b xmlt ⇒ ('a ⇒ 'b option ⇒ 'c) ⇒ 'c xmlt

```

**where**

```

xml2nd-choice tag p1 cn p2 f (XML name atts cs) = (
  let l = length cs in
  (if name = tag ∧ atts = [] ∧ l ≥ 2 then do {
    x1 ← p1 (cs ! 0);
    b ← Xmlt.bool cn (cs ! 1);
    (if b then do {
      x2 ← p2 (cs ! (l - 1));
      return (f x1 (Some x2))
    } else return (f x1 None))
 } else Xmlt.fail tag (XML name atts cs)) |
xml2nd-choice tag p1 cn p2 f xml = Xmlt.fail tag xml

```

**lemma** *xml2nd-choice-mono* [*partial-function-mono*]:

```

assumes p1:  $\bigwedge y. \text{mono-sum-bot } (p1 \ y)$ 
           $\bigwedge y. \text{mono-sum-bot } (p2 \ y)$ 
shows  $\text{mono-sum-bot } (\lambda g. \text{xml2nd-choice } t \ (\lambda y. p1 \ y \ g) \ h \ (\lambda y. p2 \ y \ g) \ f \ x)$ 
by (cases x, auto intro!: partial-function-mono p1)

```

**fun**

```

xml2to3elements ::
  string ⇒ 'a xmlt ⇒ 'b xmlt ⇒ 'c xmlt ⇒ ('a ⇒ 'b ⇒ 'c option ⇒ 'd) ⇒ 'd xmlt

```

**where**

```

xml2to3elements tag p1 p2 p3 f (XML name atts cs) = (
  let l = length cs in
  (if name = tag ∧ atts = [] ∧ l ≥ 2 ∧ l ≤ 3 then do {
    x1 ← p1 (cs ! 0);

```

```

    x2 ← p2 (cs ! 1);
    (if l = 3 then do {
      x3 ← p3 (cs ! 2);
      return (f x1 x2 (Some x3))
    } else return (f x1 x2 None))
  } else Xmlt.fail tag (XML name atts cs))) |
xml2to3elements tag p1 p2 p3 f xml = Xmlt.fail tag xml

```

**lemma** *xml2to3elements-mono* [partial-function-mono]:

**assumes** *p1*:  $\bigwedge y. \text{mono-sum-bot } (p1 \ y)$   
 $\bigwedge y. \text{mono-sum-bot } (p2 \ y)$   
 $\bigwedge y. \text{mono-sum-bot } (p3 \ y)$

**shows**  $\text{mono-sum-bot } (\lambda g. \text{xml2to3elements } t \ (\lambda y. p1 \ y \ g) \ (\lambda y. p2 \ y \ g) \ (\lambda y. p3 \ y \ g) \ f \ x)$

**by** (cases *x*, auto intro!: partial-function-mono *p1*)

**fun**

```

xml3to4elements ::
  string ⇒ 'a xmlt ⇒ 'b xmlt ⇒ 'c xmlt ⇒ 'd xmlt ⇒ ('a ⇒ 'b ⇒ 'c option ⇒
'd ⇒ 'e) ⇒
  'e xmlt

```

**where**

```

xml3to4elements tag p1 p2 p3 p4 f (XML name atts cs) = (
  let l = length cs in
  (if name = tag ∧ atts = [] ∧ l ≥ 3 ∧ l ≤ 4 then do {
    x1 ← p1 (cs ! 0);
    x2 ← p2 (cs ! 1);
    (if l = 4 then do {
      x3 ← p3 (cs ! 2);
      x4 ← p4 (cs ! 3);
      return (f x1 x2 (Some x3) x4)
    } else do {
      x4 ← p4 (cs ! 2);
      return (f x1 x2 None x4)
    } )
  } else Xmlt.fail tag (XML name atts cs))) |
xml3to4elements tag p1 p2 p3 p4 f xml = Xmlt.fail tag xml

```

**lemma** *xml3to4elements-mono* [partial-function-mono]:

**assumes** *p1*:  $\bigwedge y. \text{mono-sum-bot } (p1 \ y)$   
 $\bigwedge y. \text{mono-sum-bot } (p2 \ y)$   
 $\bigwedge y. \text{mono-sum-bot } (p3 \ y)$   
 $\bigwedge y. \text{mono-sum-bot } (p4 \ y)$

**shows**  $\text{mono-sum-bot } (\lambda g. \text{xml3to4elements } t \ (\lambda y. p1 \ y \ g) \ (\lambda y. p2 \ y \ g) \ (\lambda y. p3 \ y \ g) \ (\lambda y. p4 \ y \ g) \ f \ x)$

**by** (cases *x*, auto intro!: partial-function-mono *p1*)

**fun** *many* :: tag ⇒ 'a xmlt ⇒ ('a list ⇒ 'b) ⇒ 'b xmlt

**where**

```

many tag p f (XML name atts cs) =
  (if name = tag ∧ atts = [] then (Xmlet.map p cs ≫= (return ∘ f))
   else Xmlet.fail tag (XML name atts cs)) |
many tag p f xml = Xmlet.fail tag xml
hide-const (open) many

```

**lemma** *many-mono* [*partial-function-mono*]:  
**fixes**  $p1 :: xml \Rightarrow ('b \Rightarrow (string +_{\perp} 'c)) \Rightarrow string +_{\perp} 'd$   
**assumes**  $\bigwedge y. y \in set (Xml.children x) \Longrightarrow mono-sum-bot (p1 y)$   
**shows**  $mono-sum-bot (\lambda g. Xmlet.many t (\lambda y. p1 y g) f x)$   
**using** *assms* **by** (*cases x*) (*auto intro!*: *partial-function-mono*)

```

fun many1-gen :: tag ⇒ 'a xmlt ⇒ ('a ⇒ 'b xmlt) ⇒ ('a ⇒ 'b list ⇒ 'c) ⇒ 'c xmlt
where
  many1-gen tag p1 p2 f (XML name atts cs) =
    (if name = tag ∧ atts = [] ∧ cs ≠ [] then
      (case cs of h # t ⇒ do {
        x ← p1 h;
        xs ← Xmlet.map (p2 x) t;
        return (f x xs)
      })
     else Xmlet.fail tag (XML name atts cs)) |
  many1-gen tag p1 p2 f xml = Xmlet.fail tag xml

```

**definition** *many1* ::  $string \Rightarrow 'a xmlt \Rightarrow 'b xmlt \Rightarrow ('a \Rightarrow 'b list \Rightarrow 'c) \Rightarrow 'c xmlt$   
**where**  
 $many1\ tag\ p1\ p2 = Xmlet.many1-gen\ tag\ p1\ (\lambda -. p2)$   
**hide-const (open) many1**

**lemma** *many1-mono* [*partial-function-mono*]:  
**fixes**  $p1 :: xml \Rightarrow ('b \Rightarrow (string +_{\perp} 'c)) \Rightarrow string +_{\perp} 'd$   
**assumes**  $\bigwedge y. mono-sum-bot (p1 y)$   
**and**  $\bigwedge y. y \in set (tl (Xml.children x)) \Longrightarrow mono-sum-bot (p2 y)$   
**shows**  $mono-sum-bot (\lambda g. Xmlet.many1 t (\lambda y. p1 y g) (\lambda y. p2 y g) f x)$   
**unfolding** *Xmlet.many1-def* **using** *assms*  
**by** (*cases x, cases Xml.children x*) (*auto intro!*: *partial-function-mono*)

```

fun length-ge-2 :: 'a list ⇒ bool
where
  length-ge-2 (- # - # -) = True |
  length-ge-2 - = False

```

```

fun many2 :: tag ⇒ 'a xmlt ⇒ 'b xmlt ⇒ 'c xmlt ⇒ ('a ⇒ 'b ⇒ 'c list ⇒ 'd) ⇒
'd xmlt
where
  many2 tag p1 p2 p3 f (XML name atts cs) =
    (if name = tag ∧ atts = [] ∧ length-ge-2 cs then

```

```

    (case cs of cs0 # cs1 # t => do {
      x ← p1 cs0;
      y ← p2 cs1;
      xs ← Xmlt.map p3 t;
      return (f x y xs)
    })
  else Xmlt.fail tag (XML name atts cs) |
many2 tag p1 p2 p3 f xml = Xmlt.fail tag xml

```

**lemma** *many2-mono* [*partial-function-mono*]:  
**fixes**  $p1 :: xml \Rightarrow ('b \Rightarrow (string +_{\perp} 'c)) \Rightarrow string +_{\perp} 'd$   
**assumes**  $\bigwedge y. \text{mono-sum-bot } (p1\ y)$   
**and**  $\bigwedge y. \text{mono-sum-bot } (p2\ y)$   
**and**  $\bigwedge y. \text{mono-sum-bot } (p3\ y)$   
**shows**  $\text{mono-sum-bot } (\lambda g. Xmlt.many2\ t\ (\lambda y. p1\ y\ g)\ (\lambda y. p2\ y\ g)\ (\lambda y. p3\ y\ g)\ f\ x)$   
**using** *assms*  
**by** (*cases x, cases Xml.children x, (auto intro!: partial-function-mono)[1], cases tl (Xml.children x), auto intro!: partial-function-mono*)

**fun**  
*xml1or2many-elements* ::  
 $string \Rightarrow 'a\ xmlt \Rightarrow 'b\ xmlt \Rightarrow 'c\ xmlt \Rightarrow ('a \Rightarrow 'b\ option \Rightarrow 'c\ list \Rightarrow 'd) \Rightarrow 'd\ xmlt$   
**where**  
*xml1or2many-elements tag p1 p2 p3 f (XML name atts cs) =*  
*(if name = tag  $\wedge$  atts = []  $\wedge$  cs  $\neq$  [] then*  
*(case cs of*  
*cs0 # tt =>*  
*do {*  
*x ← p1 cs0;*  
*(case tt of*  
*cs1 # t =>*  
*do {*  
*try do {*  
*y ← p2 cs1;*  
*xs ← Xmlt.map p3 t;*  
*return (f x (Some y) xs)*  
*} catch (\ -. do {*  
*xs ← Xmlt.map p3 tt;*  
*return (f x None xs)*  
*})*  
*}*  
*| [] => return (f x None []))}*  
*else Xmlt.fail tag (XML name atts cs) |*  
*xml1or2many-elements tag p1 p2 p3 f xml = Xmlt.fail tag xml*

**fun**  
*xml1many2elements-gen* ::



$string \Rightarrow 'a \text{ xmlt} \Rightarrow ('a \Rightarrow 'b \text{ xmlt}) \Rightarrow 'c \text{ xmlt} \Rightarrow 'd \text{ xmlt} \Rightarrow$   
 $('a \Rightarrow 'b \text{ list} \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e) \Rightarrow 'e \text{ xmlt}$

**where**

```

xml1many2elements-gen tag p1 p2 p3 p4 f (XML name atts cs) = (
  let ds = List.rev cs; l = length cs in
  (if name = tag ^ atts = [] ^ l ≥ 3 then do {
    x ← p1 (cs ! 0);
    xs ← Xmlt.map (p2 x) (tl (take (l - 2) cs));
    y ← p3 (ds ! 1);
    z ← p4 (ds ! 0);
    return (f x xs y z)
  } else Xmlt.fail tag (XML name atts cs))) |
xml1many2elements-gen tag p1 p2 p3 p4 f xml = Xmlt.fail tag xml

```

**lemma** *xml1many2elements-gen-mono* [partial-function-mono]:

**fixes**  $p1 :: \text{xml} \Rightarrow ('b \Rightarrow (\text{string} +_{\perp} 'c)) \Rightarrow \text{string} +_{\perp} 'd$

**assumes**  $p1: \bigwedge y. \text{mono-sum-bot} (p1 y)$

$\bigwedge y. \text{mono-sum-bot} (p3 y)$

$\bigwedge y. \text{mono-sum-bot} (p4 y)$

**shows**  $\text{mono-sum-bot} (\lambda g. \text{xml1many2elements-gen } t (\lambda y. p1 y g) p2 (\lambda y. p3 y g) (\lambda y. p4 y g) f x)$

**by** (cases x, auto intro!: partial-function-mono p1)

**fun**

*xml1many2elements* ::

$string \Rightarrow 'a \text{ xmlt} \Rightarrow 'b \text{ xmlt} \Rightarrow 'c \text{ xmlt} \Rightarrow 'd \text{ xmlt} \Rightarrow ('a \Rightarrow 'b \text{ list} \Rightarrow 'c \Rightarrow 'd \Rightarrow$   
 $'e) \Rightarrow$   
 $'e \text{ xmlt}$

**where**

$\text{xml1many2elements tag p1 p2} = \text{xml1many2elements-gen tag p1 } (\lambda-. p2)$

**fun**

*xml-many2elements* ::

$string \Rightarrow 'a \text{ xmlt} \Rightarrow 'b \text{ xmlt} \Rightarrow 'c \text{ xmlt} \Rightarrow ('a \text{ list} \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'd \text{ xmlt}$

**where**

```

xml-many2elements tag p1 p2 p3 f (XML name atts cs) = (

```

```

  let ds = List.rev cs in

```

```

  (if name = tag ^ atts = [] ^ length-ge-2 cs then do {

```

```

    xs ← Xmlt.map p1 (List.rev (tl (tl ds)));

```

```

    y ← p2 (ds ! 1);

```

```

    z ← p3 (ds ! 0);

```

```

    return (f xs y z)

```

```

  } else Xmlt.fail tag (XML name atts cs))) |

```

```

xml-many2elements tag p1 p2 p3 f xml = Xmlt.fail tag xml

```

**definition** *options* ::  $(\text{string} \times 'a \text{ xmlt}) \text{ list} \Rightarrow 'a \text{ xmlt}$

**where**

*options ps x* =

(case map-of ps (Xml.tag x) of

```

      None  $\Rightarrow$  error (concat
        ["expected one of: ", concat (map ( $\lambda p$ . fst p @ " ") ps), "↔", "but found",
        "↔", show x])
      | Some p  $\Rightarrow$  p x
hide-const (open) options

```

```

lemma options-mono-gen [partial-function-mono]:
  assumes p:  $\bigwedge k p. (k, p) \in \text{set } ps \implies \text{mono-sum-bot } (p \ x)$ 
  shows mono-sum-bot ( $\lambda g. \text{Xmlt.options } (\text{map } (\lambda (k, p). (k, (\lambda y. p \ y \ g))) \ ps) \ x$ )
proof -
  {
    fix g
    have ( $\text{map } (\lambda p. \text{fst } p \ @ \ " \ ") (\text{map } (\lambda (k, p). (k, \lambda y. p \ y \ g)) \ ps)$ ) =
       $\text{map } (\lambda p. \text{fst } p \ @ \ " \ ") \ ps$ 
    by (induct ps) (auto)
  } note id = this
  {
    fix z
    have mono-sum-bot
      ( $\lambda g. \text{case map-of } (\text{map } (\lambda (k, p). (k, \lambda y. p \ y \ g)) \ ps) \ (\text{Xml.tag } x) \ \text{of}$ 
        None  $\Rightarrow$  z
        | Some p  $\Rightarrow$  p x)
    using p
    proof (induct ps)
      case Nil
      show ?case by (auto intro: partial-function-mono)
    next
      case (Cons kp ps)
      obtain k p where kp: kp = (k,p) by force
      note Cons = Cons[unfolded kp]
      from Cons(2) have monop: mono-sum-bot (p x) and mono:  $\bigwedge k p. (k,p) \in$ 
        set ps  $\implies$  mono-sum-bot (p x) by auto
      show ?case
      proof (cases Xml.tag x = k)
        case True
        thus ?thesis unfolding kp using monop by auto
      next
        case False
        thus ?thesis using Cons(1) mono unfolding kp by auto
      qed
    qed
  } note main = this
  show ?thesis unfolding Xmlt.options-def
  unfolding id
  by (rule main)
qed

```

```

local-setup  $\langle \text{fn } lthy \Rightarrow$ 

```

```

let
  val N = 30 (* we require monotonicity lemmas for xml-options for lists up to
length N *)
  val lthy = Proof-Context.theory-of lthy
  val options = @ {term Xmlt.options :: (string × (xml ⇒ (string +⊥ 'd))) list ⇒
xml ⇒ string +⊥ 'd}
  val mono-sum-bot = @ {term mono-sum-bot :: (('a ⇒ ('b +⊥ 'c)) ⇒ string +⊥
'd) ⇒ bool}
  val ktyp = @ {typ string}
  val x = @ {term x :: xml}
  val y = @ {term y :: xml}
  val g = @ {term g :: 'a ⇒ 'b +⊥ 'c}
  val ptyp = @ {typ xml ⇒ ('a ⇒ ('b +⊥ 'c)) ⇒ string +⊥ 'd}
  fun k i = Free (k ^ string-of-int i, ktyp)
  fun p i = Free (p ^ string-of-int i, ptyp)
  fun prem i = HOLogic.mk-Trueprop (mono-sum-bot $ (p i $ x))
  fun prems n = 1 upto n |> map prem
  fun pair i = HOLogic.mk-prod (k i, lambda y (p i $ y $ g))
  fun pair2 i = HOLogic.mk-prod (k i, p i)
  fun list n = 1 upto n |> map pair |> HOLogic.mk-list @ {typ (string × (xml ⇒
string +⊥ 'd))}
  fun list2 n = 1 upto n |> map pair2 |> HOLogic.mk-list (HOLogic.mk-prodT
(ktyp, ptyp))
  fun concl n = HOLogic.mk-Trueprop (mono-sum-bot $ lambda g (options $ (list
n) $ x))
  fun xs n = x :: (1 upto n |> map (fn i => [p i, k i]) |> List.concat)
|> map (fst o dest-Free)
  fun tac n pc =
    let
      val {prems = prems, context = ctxt} = pc
      val mono-thm = Thm.instantiate'
        (map (SOME o Thm.ctyp-of ctxt) [@ {typ 'a'}, @ {typ 'b'}, @ {typ 'c'}, @ {typ
'd}])
        (map (SOME o Thm.cterm-of ctxt) [list2 n, x]) @ {thm Xmlt.options-mono-gen}
    in
      Method.insert-tac ctxt (mono-thm :: prems) 1 THEN force-tac ctxt 1
    end
  fun thm n = Goal.prove lthy (xs n) (prems n) (concl n) (tac n)
  val thms = map thm (0 upto N)
in Local-Theory.note ((@ {binding options-mono-thms}, []), thms) lthy |> snd end
)

```

```

declare Xmlt.options-mono-thms [partial-function-mono]

```

```

fun choice :: string ⇒ 'a xmlt list ⇒ 'a xmlt

```

```

where

```

```

  choice e [] x = error (concat ["error in parsing choice for ", e, "⊠", show x]) |
  choice e (p # ps) x = (try p x catch (λ-. choice e ps x))

```

```

hide-const (open) choice

```

**lemma** *choice-mono-2* [*partial-function-mono*]:

**assumes** *p*: *mono-sum-bot* (*p1* *x*)  
*mono-sum-bot* (*p2* *x*)

**shows** *mono-sum-bot* ( $\lambda g. \text{Xmlet.choice } e \ [(\lambda y. p1\ y\ g), (\lambda y. p2\ y\ g)]\ x$ )

**using** *p* **by** (*auto intro!*: *partial-function-mono*)

**lemma** *choice-mono-3* [*partial-function-mono*]:

**assumes** *p*: *mono-sum-bot* (*p1* *x*)  
*mono-sum-bot* (*p2* *x*)  
*mono-sum-bot* (*p3* *x*)

**shows** *mono-sum-bot* ( $\lambda g. \text{Xmlet.choice } e \ [(\lambda y. p1\ y\ g), (\lambda y. p2\ y\ g), (\lambda y. p3\ y\ g)]\ x$ )

**using** *p* **by** (*auto intro!*: *partial-function-mono*)

**fun** *change* :: 'a *xmlet*  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'b *xmlet*

**where**

*change* *p* *f* *x* = *p* *x*  $\gg=$  *return*  $\circ$  *f*

**hide-const** (**open**) *change*

**lemma** *change-mono* [*partial-function-mono*]:

**assumes** *p*:  $\bigwedge y. \text{mono-sum-bot } (p1\ y)$

**shows** *mono-sum-bot* ( $\lambda g. \text{Xmlet.change } (\lambda y. p1\ y\ g)\ f\ x$ )

**by** (*cases* *x*, *insert* *p*, *auto intro!*: *partial-function-mono*)

**fun** *int-of-digit* :: *char*  $\Rightarrow$  *string*  $+_{\perp}$  *int*

**where**

*int-of-digit* *x* =

(if *x* = *CHR* "0" then return 0

else if *x* = *CHR* "1" then return 1

else if *x* = *CHR* "2" then return 2

else if *x* = *CHR* "3" then return 3

else if *x* = *CHR* "4" then return 4

else if *x* = *CHR* "5" then return 5

else if *x* = *CHR* "6" then return 6

else if *x* = *CHR* "7" then return 7

else if *x* = *CHR* "8" then return 8

else if *x* = *CHR* "9" then return 9

else error (*x* # " is not a digit")

**fun** *int-of-string-aux* :: *int*  $\Rightarrow$  *string*  $\Rightarrow$  *string*  $+_{\perp}$  *int*

**where**

*int-of-string-aux* *n* [] = return *n* |

*int-of-string-aux* *n* (*d* # *s*) = (*int-of-digit* *d*  $\gg=$  ( $\lambda m. \text{int-of-string-aux } (10 * n + m)\ s$ ))

**definition** *int-of-string* :: *string*  $\Rightarrow$  *string*  $+_{\perp}$  *int*

**where**

*int-of-string* *s* =

```

    (if s = [] then error "cannot convert empty string into number"
     else if take 1 s = "-" then int-of-string-aux 0 (tl s) >>= (λ i. return (0 - i))
     else int-of-string-aux 0 s)

```

**hide-const** *int-of-string-aux*

```

fun int :: tag ⇒ int xmlt
where
  int tag x = (Xmlt.text tag x >>= int-of-string)
hide-const (open) int

```

```

fun nat :: tag ⇒ nat xmlt
where
  nat tag x = do {
    txt ← Xmlt.text tag x;
    i ← int-of-string txt;
    return (Int.nat i)
  }
hide-const (open) nat

```

```

definition rat :: rat xmlt
where
  rat = Xmlt.options [
    ("integer", Xmlt.change (Xmlt.int "integer") of-int),
    ("rational",
     Xmlt.pair "rational" (Xmlt.int "numerator") (Xmlt.int "denominator")
     (λ x y. of-int x / of-int y))]
hide-const (open) rat

```

**end**